DOCUMENT RESUME

ED 274 326 IR 012 306

AUTHOR	Cugini, John V.
TITLE	Selection and Use of General-Purpose Programming Languages--Overview. Volume 1.
INSTITUTION	National Bureau of Standards (DOC), hington, D.C. Inst. for Computer Sciences and Techn ogy.
REPORT NO	NBS-SP-500-117/1
PUB DATE	Oct 84
NOTE	100p.
AVAILABLE FROM	Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402.
PUB TYPE	Guides - General (050) -- Information Analyses (070)
EDRS PRICE	MF01/PC04 Plus Postage.
DESCRIPTORS	Authoring Aids (Programing); *Comparative Analysis; Computer Graphics; *Computer Oriented Programs; Database Management Systems; Data Processing; *Evaluation Criteria; Federal Government; Industry; *Microcomputers; Programing; *Programing Languages; Selection; *Standards
IDENTIFIERS	BASIC Programing Language; COBOL Programing Language; FORTRAN Programing Language; PASCAL Programing Language

ABSTRACT

This study presents a review of selection factors for the seven major general-purpose programming languages: Ada, BASIC, C, COBOL, FORTRAN, PASCAL, and PL/I. The factors covered include not only the logical operations within each language, but also the advantages and disadvantages stemming from the current computing environment, e.g., software packages, microcomputers, and standards. This volume contains the discussion of language selection criteria based on: (1) the language and its implementation; (2) the application to be programmed; and (3) the user's existing facilities and software. Explanations of the criteria associated with the application and the user's facilities are included as well as 49 references. Appendices contain a list of abbreviations used, an annotated list of organizations that can provide additional information, and a discussion of alternatives to conventional programming. (Author/DJR)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. Department
of Commerce

National Bureau
of Standards

Computer Science and Technology

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

X This document has been reproduced as received from the person or organization originating it.

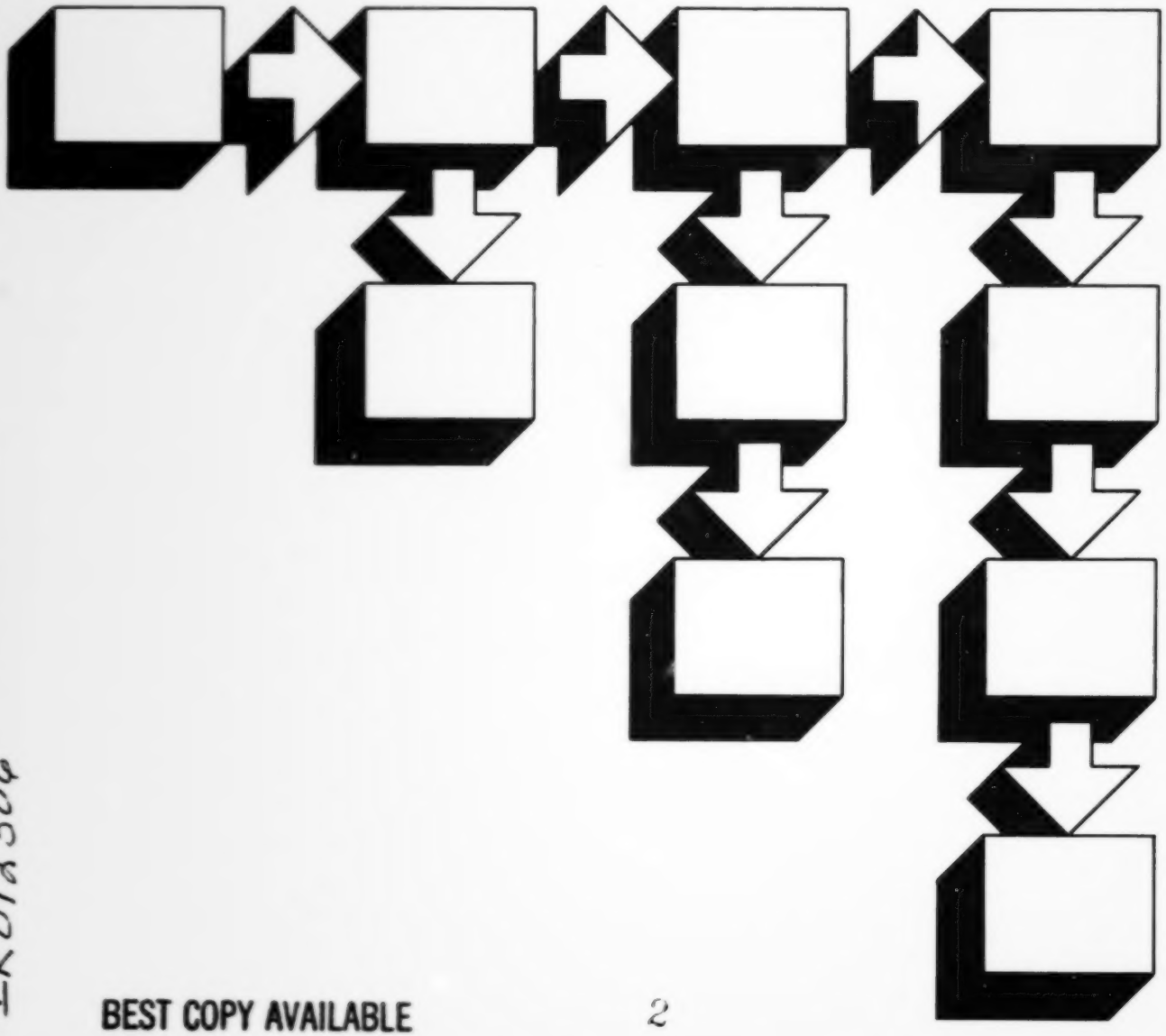
☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED274326

NBS Special Publication 500-117, Volume 1

Selection and Use of General-Purpose Programming Languages — Overview



IR012304

BEST COPY AVAILABLE

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, the Institute for Computer Sciences and Technology, and the Center for Materials Science.

The National Measurement Laboratory

Provides the national system of physical and chemical measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; provides advisory and research services to other Government agencies; conducts physical and chemical research; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

- Basic Standards[2]
- Radiation Research
- Chemical Physics
- Analytical Chemistry

The National Engineering Laboratory

Provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

- Applied Mathematics
- Electronics and Electrical Engineering[2]
- Manufacturing Engineering
- Building Technology
- Fire Research
- Chemical Engineering[2]

The Institute for Computer Sciences and Technology

Conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

- Programming Science and Technology
- Computer Systems Engineering

The Center for Materials Science

Conducts research and provides measurements, data, standards, reference materials, quantitative understanding and other technical information fundamental to the processing, structure, properties and performance of materials; addresses the scientific basis for new advanced materials technologies; plans research around cross-country scientific themes such as nondestructive evaluation and phase diagram development; oversees Bureau-wide technical programs in nuclear reactor radiation research and nondestructive evaluation; and broadly disseminates generic technical information resulting from its programs. The Center consists of the following Divisions:

- Inorganic Materials
- Fracture and Deformation[3]
- Polymers
- Metallurgy
- Reactor Radiation

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Gaithersburg, MD 20899.
[2]Some divisions within the center are located at Boulder, CO 80303.
[3]Located at Boulder, CO, with some elements at Gaithersburg, MD.

Computer Science and Technology

NBS Special Publication 500-117, Volume 1

Selection and Use of General-Purpose Programming Languages — Overview

John V. Cugini

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued October 1984

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

Library of Congress Catalog Card Number: 84-601119

National Bureau of Standards Special Publication 500-117, Volume 1
Natl. Bur. Stand. (U.S.), Spec. Publ. 500-117, Vol. 1, 81 pages (Oct. 1984)
CODEN: XNBSAV

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1984

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402

PREFACE: Role of ICST

The Institute for Computer Sciences and Technology (ICST) within the National Bureau of Standards (NBS) has a mission under Public Law 89-306 (Brooks Act) to promote the "economic and efficient purchase, lease, maintenance, operation, and utilization of automatic data processing equipment by Federal departments and agencies." Thus, ICST pursues a number of different approaches to the problem of application development and maintenance. When a potentially valuable technique first appears, ICST may be involved in research and evaluation. Later on, standardization of the results of such research, in cooperation with voluntary industry standards bodies, may best serve Federal interests. Finally, ICST helps Federal agencies make practical use of existing standards and technology through direct consulting and the development of supporting guidelines and software.

The development and promotion of standard programming languages provide an especially clear example of this cycle of technological development. Through its activities within the Conference on Data System Languages (CODASYL), the Institute of Electrical and Electronics Engineers Computer Society (IEEE/CS), the International Standards Organization (ISO), and committees accredited by the American National Standards Institute (ANSI), ICST has contributed to the design or standardization of most of the prominent languages in use today.

Technical work within such organizations helps to promote good language design. ICST represents Federal users' interests by striving for the inclusion of language features which exploit advances in programming technology and software engineering.

Beyond the advantages of a well-designed language, standardization per se is valuable for several reasons. First and foremost, good language standards make it easier and less costly to transport software from one language processor to another, either within the systems of a given vendor, or between vendors. This capability is valuable not only for programs written by end-users, but also encourages the development of vendor-independent commercial software. Second, standards help to preserve the value of programmers' skills as they move from one installation to another. There is no need for extensive retraining in local dialects of COBOL, for instance. When a need arises for hiring, there exists a large pool of programmers knowledgable in the language. Third, programming language standards allow the development of standard language bindings to various application facilities, such as graphics, communications, or database. Finally, standards provide stability to the definition of a language. Thus, the large base of Federal software is not threatened by arbitrary changes in language implementation. Changes are introduced in a controlled way and only after careful evaluation of the effect on existing programs.

Thus, ICST aims to achieve a reasonable balance between the incorporation of improved techniques for software development and maintenance on the one hand, and the protection of existing applications on the other. Given the size of Federal data processing (DP) operations, the achievement of such a balance is a critical task. Billions of dollars are at stake, both in ongoing development and maintenance activities and in the base of existing software. The challenge is to take advantage of potential savings in the former while minimizing costs for the latter.

When ICST determines that a language can provide important benefits for Federal users, and that a technically sound specification exists, it recommends the language to the Secretary of Commerce for adoption as a Federal Information Processing Standard (FIPS) [NBS75], [NBS80], [NBS80a]. The reasons for accepting a language as the subject of a FIPS are based on much the same criteria as described below for language selection. The nature of the language, the applications typical of Federal users, and the existing base of programs, machines, language processors, and programming skills are all taken into account.

When a FIPS is issued, the Federal users of that standard then receive a number of support services. They may request official Federal interpretations from ICST as to the meaning of the standard if a question arises about a particular implementation [NBS81]. These interpretations are developed in cooperation with language experts in Federal agencies. ICST works closely with the General Services Admininstration's (GSA) Federal Software Testing Center (FSTC), which is responsible for the validation of language processors claiming to conform to the FIPS. FSTC maintains a list of certified language processors [FSTC84] which have undergone validation. Finally, ICST participates in national and international standards activities for the FIPS languages and stands ready to assist agencies with various language issues, such as the applicability of a language, technical questions about the meaning of a standard, and information on the likely development path for a given language standard.

Selection and Use of General-Purpose Programming Languages
Volume 1 - Overview

John V. Cugini
Institute for Computer Sciences and Technology
National Bureau of Standards

ABSTRACT

Programming languages have been and will continue to be an important instrument for the automation of a wide variety of functions within industry and the Federal Government. Other instruments, such as program generators, application packages, query languages, and the like, are also available and their use is preferable in some circumstances.

Given that conventional programming is the appropriate technique for a particular application, the choice among the various languages becomes an important issue. There are a great number of selection criteria, not all of which depend directly on the language itself. Broadly speaking, the criteria are based on 1) the language and its implementation, 2) the application to be programmed, and 3) the user's existing facilities and software.

This study presents a survey of selection factors for the major general-purpose languages: Ada*, BASIC, C, COBOL, FORTRAN, Pascal, and PL/I. The factors covered include not only the logical operations within each language, but also the advantages and disadvantages stemming from the current computing environment, e.g., software packages, microcomputers, and standards. The criteria associated with the application and the user's facilities are explained. Finally, there is a set of program examples to illustrate the features of the various languages.

This volume contains the discussion of language selection criteria. Volume 2 comprises the program examples.

Key words: Ada; alternatives to programming; BASIC; C; COBOL; FORTRAN; Pascal; PL/I; programming language features; programming languages; selection of programming language.

* Ada is a registered trademark of the U. S. Government, Ada Joint Project Office.

TABLE OF CONTENTS: Volume 1 - Overview

1.0 PURPOSE AND SCOPE 1
2.0 PROGRAMMING LANGUAGES - CRITERIA AND COMPARISON . . . 1
2.1 Language Factors 2
2.1.1 Syntactic Style (see Figure 1) 4
2.1.1.1 Statement Terminator 4
2.1.1.2 Fixed Or Free Format 6
2.1.1.3 Statement Labels 6
2.1.1.4 Identifiers 6
2.1.1.5 Implicit Or Declared Entities 6
2.1.1.6 Program Length 7
2.1.2 Semantic Structure 7
2.1.2.1 Control Of Execution (see Figure 2) 7
2.1.2.1.1 Structured Programming 9
2.1.2.1.2 Blocks 9
2.1.2.1.3 Subroutines 9
2.1.2.1.4 Functions 10
2.1.2.1.5 Recursion 10
2.1.2.1.6 Generic Procedures 10
2.1.2.1.7 Exception Handling 11
2.1.2.1.8 Concurrency 11
2.1.2.2 Control Of Data (see Figure 3) 12
2.1.2.2.1 Storage Classes 12
2.1.2.2.2 External Data 14
2.1.2.2.3 Data Abstraction 14
2.1.2.3 Packages 15
2.1.3 Data Types And Manipulation 15
2.1.3.1 Checking And Coercion 15
2.1.3.2 Elementary Data 16
2.1.3.2.1 Numeric (see Figure 4) 16
2.1.3.2.2 Character (see Figure 5) 18
2.1.3.2.3 Logical (see Figure 6) 18
2.1.3.2.4 Bit (see Figure 7) 21
2.1.3.2.5 Pointer (see Figure 8) 21
2.1.3.3 Aggregate Data 21
2.1.3.3.1 Arrays (see Figure 9) 23
2.1.3.3.2 Files And I/O (see Figure 10) 25
2.1.3.3.3 Records (see Figure 11) 27
2.1.3.3.4 Sets (see Figure 12) 29
2.1.4 Application Facilities (see Figure 13) 29
2.1.4.1 Reports 29
2.1.4.2 Database 32
2.1.4.3 Real-time 32
2.1.4.4 Communication 32
2.1.4.5 Graphics 33

2.1.5 Program Implementation Control 33
2.1.6 Simplicity 34
2.1.7 Standardization (see Figure 14) 35
2.1.8 Performance 37
2.1.9 Software Availability 38
2.1.10 Software Development Support 39
2.1.10.1 Source Code Manipulation And Checking 40
2.1.10.2 Program Testing 41
2.1.10.3 Information And Analysis 42
2.2 Application Requirements (see Figures 15 and 16) . 42
2.2.1 Functional Operations 42
2.2.2 Size And Complexity 45
2.2.3 Number Of Programmers 46
2.2.4 Expertise 46
2.2.5 End-user Interaction 47
2.2.6 Reliability 47
2.2.7 Timeframe 48
2.2.8 Portability 48
2.2.9 Execution Efficiency 48
2.3 Installation Requirements 49
2.3.1 Language Availability 49
2.3.2 Compatibility With Existing Software 50
3.0 LANGUAGE SUMMARY (see Figure 17) 51
3.1 Ada 51
3.2 BASIC 51
3.3 C 53
3.4 COBOL 53
3.5 FORTRAN 54
3.6 Pascal 55
3.7 PL/I 55
4.0 CONCLUSION 56

REFERENCES 57

ACKNOWLEDGMENTS 62

APPENDIX A ABBREVIATIONS

APPENDIX B SOURCES OF INFORMATION

B.1 INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY . . . B-1
B.2 FEDERAL SOFTWARE TESTING CENTER B-1
B.3 NATIONAL TECHNICAL INFORMATION SERVICE B-2
B.4 X3 - INFORMATION AND PROCESSING SYSTEMS B-2
B.5 SC5 - PROGRAMMING LANGUAGES B-2
B.6 IEEE COMPUTER SOCIETY B-3
B.7 SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES . . B-3

APPENDIX C ALTERNATIVES TO CONVENTIONAL PROGRAMMING

C.1 DATABASE MANAGEMENT SYSTEMS C-3
C.2 QUERY AND REPORT FACILITIES C-3
C.3 APPLICATION PACKAGES C-3
C.4 APPLICATION GENERATORS C-3
C.5 VERY HIGH-LEVEL LANGUAGES C-4
C.6 ASSEMBLER LANGUAGE C-4
C.7 MANUAL OPERATIONS C-4

FIGURES:

Figure 1 - Syntactic Style 5
Figure 2 - Control of Execution 8
Figure 3 - Control of Data 13
Figure 4 - Numeric Data and Manipulation 17
Figure 5 - Character Data and Manipulation 19
Figure 6 - Logical Data and Manipulation 20
Figure 7 - Bit Data and Manipulation 22
Figure 8 - Pointer Data and Manipulation 22
Figure 9 - Arrays 24
Figure 10 - Files and I/O 26
Figure 11 - Records 28
Figure 12 - Sets 30
Figure 13 - Application Facilities 31
Figure 14 - Standardization 36
Figure 15 - Language Factors vs. Application Requirements . 43
Figure 16 - Languages vs. Application Requirements 44
Figure 17 - Languages and Standards Bodies 52

1.0 PURPOSE AND SCOPE

Programming is a means to an end. In this report, we shall assume that the end is the automation of some function performed by an individual or an organization, such as a company or Federal agency. The question, then, is "which programming language is best for a given application?" Any discussion of programming and programming languages must consider them within the general context of data processing. It is not enough to know the logical structure of the various languages. In order to make informed choices, we must also take into account such factors as portability, the availability of languages in various computing environments (e.g., main-frames vs. micros), the availability of software for the language, and so on. The purpose of this report is to present and explain a set of language selection criteria which DP managers and users may apply to their particular situations. These criteria apply, of course, only when the language of implementation is important to the user. Conversely, if the user is purchasing a software package to be used strictly as is, the language in which the package is written may be of no concern.

For this report, the scope of consideration shall be limited to conventional programming languages as the means for implementing and maintaining an application system. It is important to understand that programming is only one among many application development techniques. Some of these alternative techniques are described in Appendix C. Although this report focuses on issues of language use and selection, it is by no means implied that such alternatives are to be ruled out in favor of a conventional programming approach.

2.0 PROGRAMMING LANGUAGES - CRITERIA AND COMPARISON

Choosing the appropriate language is a difficult process because there are such a large number of relevant factors. The purpose of this section is twofold: first to enumerate the most significant of those factors, and second to organize them in such a way that the relationships among them may be understood. We will group language selection criteria into three broad categories: 1) the properties of the various languages and of their associated software, 2) the nature of the application being programmed, and 3) the characteristics of the installation involved in the work. The first set of criteria, based on language properties, explain the features offered by the different languages. These features must then be evaluated against the requirements imposed by the application to be programmed and the characteristics of the installation.

Language and application issues encompass both logical and practical considerations. Logical properties are those that are true by definition of the object in question: COBOL is defined to have fixed-point decimal arithmetic; the specification of a

payroll system requires that one of its functions is to compute time-and-a-half for overtime. Conversely, it is a practical consideration that COBOL is more widely available than SNOBOL4 or that the payroll system will be run on three different vendors' machines. All installation characteristics are assumed to be practical. Within a category, logical criteria will be discussed first, and then the practical criteria.

This distinction between logical and practical considerations is important, since much of the literature on language usage covers only the logical criteria. This is understandable, in that the matching of the inherent properties of a language with the logical definition of an application is rightly seen as a central part of the selection process. Nonetheless, DP managers operate under "real-world" constraints, and even though a language may be a theoretically perfect match for an application, there may be mundane but effective reasons (e.g., none of the programmers knows the language, the language is unavailable on the installation's hardware) for making another choice.

Although this report lists and explains the various criteria to be taken into account, it is up to each organization to decide which are most important. For any given application, it is unlikely that all the criteria will favor one language. When weighing conflicting factors, one should evaluate both long-term and short-term costs and benefits. For instance, changing from one language to another will generate costs in the short term, but whether these costs are justified depends on the prospects for ongoing savings.

This report will cover Ada*, BASIC, C, COBOL, FORTRAN, Pascal, and PL/I. These languages were chosen because they are currently the most used by Federal agencies, or are likely to become widely used. We recognize that there are other languages (ALGOL, APL, FORTH, LISP, MODULA, MUMPS, PROLOG, SNOBOL, etc.) with certain advantages, but these are either oriented to some special application area or in less common use and so they are not included here. The seven languages under study are not all approved FIPS. Please see the preface and section 2.1.7 for details on the role of FIPS and the status of standards for these languages.

2.1 Language Factors

This section will compare various languages, first with respect to their syntactic and semantic features, and then with respect to implementation and environmental issues. We will describe each language according to its definition in the corresponding ANSI standard ([Ada83], [COBO74], [FORT78],

* Ada is a registered trademark of the U. S. Government, Ada Joint Project Office.

[Pasc83], and [PL/I76]). In the case of BASIC and C, a comprehensive language standard does not yet exist. The base documents for these languages are given in [BASI84] and [Kern78] (see section 2.1.7).

There is an ANSI standard and a FIPS for Minimal BASIC, a subset of the language considered throughout this report. Although many implementations conform to this standard, the language it describes is so small that there are several implementor-defined enhancements. For C, we shall assume that the standard input/output (I/O) library is available, but not the UNIX* interface. A new version of COBOL is currently in the approval process [COBO83], and its enhancements over the current standard are noted. Where it is necessary to distinguish, we shall refer to these two versions as "COBOL-74" and "COBOL-8x". Otherwise, the term "COBOL" may be assumed to apply to both versions. Several of the language definitions have a number of levels or subsets [PL/I81]. In general, we shall compare the most complete versions which are defined, and disregard lower levels and subsets.

Bear in mind, then, that the status of the languages under scrutiny varies considerably. There are as yet few implementations of and little experience with the language specifications for Ada and BASIC. At the other extreme, COBOL and FORTRAN assumed their present shape years ago, and their advantages and limitations are by now apparent. Although draft standards exist for BASIC and C, these have not been formally adopted, as is the case for the other languages.

It should be noted that we are comparing features as directly supported in the language. Clearly, most features can be simulated with a greater or lesser degree of effort. Thus, the absence of a logical data-type in BASIC, for instance, does not prevent a programmer from setting up a character variable as a switch and assigning "T" or "F" to it. Nonetheless, this puts the burden on the user, rather than the language. In such cases, then, we shall simply say that BASIC does not support logical data, without intending to preclude the possibility of achieving the same effect some other way.

Another point to keep clear is that we shall be concerned with the facilities guaranteed to the user by standard-conforming implementations of the language, whether or not these facilities are built directly into the syntax of the language, or provided indirectly, via standard runtime support. For our purposes then, we shall simply say that both Ada and COBOL provide sequential files, even though they are provided by means of predefined generic packages in Ada and directly in the syntax of COBOL. Conversely, we shall say that Ada does not "have" keyed files, because even though there could be a package to support them, no such package is required by the standard.

* UNIX is a trademark of Bell Laboratories.

This survey is intended to convey the general capabilities of each of the languages. A few specialized features (e.g., the label data type in PL/I) have been omitted in the interest of brevity.

This report takes a comparative approach when discussing the languages: they are measured against each other rather than against an abstract ideal. Thus, we shall emphasize the points at which a given language differs from the prevailing pattern. Generally, each section first describes the capability under consideration, then notes the typical treatment (if any) of that capability, and finally points out exceptions to the typical case. Many of the sections have an associated figure. These figures normally should not be used in isolation from the accompanying text. Their purpose is to summarize the discussion and serve as a reminder of which language has which feature; by themselves, they may not convey fully accurate information.

Unfortunately, there is a great disparity in the terminology used by the various language communities to describe similar concepts. There is no consistent usage for terms such as "block", "procedure", "identifier", or "name", e.g. In this guide we have adopted the usage judged most prevalent. In those sections where confusion is likely, the discussion is prefaced by a brief, informal definition of the concept in question. This definition is not meant to be authoritative, but merely serves to avoid ambiguity.

2.1.1 Syntactic Style (see Figure 1) -

This category includes those features of the language which determine the general appearance of the program, but have no direct bearing on the ability to express control or data structures.

2.1.1.1 Statement Terminator -

A statement in a language describes a single action or object. Of the seven languages, only FORTRAN and BASIC use an end-of-line to mark the end of a statement. For the other languages, lines are not logically significant. In COBOL, the beginning of a statement is denoted by a keyword and a sequence of statements, called a sentence, is terminated by a period (".") . All the others use a semicolon (";") to delimit statements. Nonetheless, all languages generally encourage the convention of coding one statement per line. The rules for using semicolons, especially within compound statements, are sometimes confusing to novice programmers and so the ability to dispense with them favors ease of writing.

Figure 1 - Syntactic Style (see section 2.1.1)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Statement Terminator	";"	end-of-line	";"	new verb or "."	end-of-line	";"*	";"
Free Format	Yes	Partial	Yes	No	No	Yes	Yes
Labels	Name	Number	Name	Name	Number	Number	Name
Identifier Size (maximum)	any size	31	8	30	6	any size	31
Reserved Words	Yes	Yes	Yes	Yes	No	Yes	No
Undeclared Variables	No	Yes	No	No	Yes	No	Yes
Program Length (overall style)	Long	Short	Medium	Long	Short	Medium	Long

* separator, not a terminator

2.1.1.2 Fixed Or Free Format -

COBOL and FORTRAN both have conventions about which character position in a line of source code must be used for certain syntactic entities, such as labels, continuation, or the start of a statement. In BASIC, every line must begin with a line number, starting in the first position. Certainly it is desirable that programs be written with some convention for indenting; whether these conventions should be part of the language is questionable, but there should be some mechanism for enforcing an orderly style. Language-based editors and prettyprinters may be used for this purpose.

2.1.1.3 Statement Labels -

FORTRAN, BASIC, and Pascal use numbers as statement labels. Clearly, it is preferable to be able to name, rather than number, locations within the code, as the other languages allow.

2.1.1.4 Identifiers -

Identifiers (often called names) are syntactic objects used to denote various kinds of entities such as data, types, and procedures. Only FORTRAN and C still limit identifiers to a small number of characters (six and eight, respectively). This is also a problem with many small versions of BASIC, some of which limit the user to a mere two characters, but the proposed standard will allow 31 characters. Either for reading or writing, a low limit on the length of identifiers is a severe hindrance to good programming. Some implementations allow a large number of characters but only the few first are significant. For instance, C will accept more than eight characters, but may not recognize these additional characters. This approach can be deceptive and counterproductive because entities that appear to be different in the source code may actually be the same.

There may be other restrictions on identifiers. Most of the languages have a lengthy list of reserved words which must not be used as identifiers. This restriction can cause confusion, especially among inexperienced programmers. FORTRAN and PL/I have no reserved words, and BASIC only a few.

2.1.1.5 Implicit Or Declared Entities -

Here is another case where ease of reading and of writing tend to be opposed. Clearly, when writing it is easier to be able to assume the existence of entities, such as variables, as they are needed in the algorithm. This is allowed in FORTRAN,

BASIC, and PL/I. On the other hand, by requiring a program explicitly to declare all variables before they are referenced, COBOL, Ada, Pascal, and C enforce a certain discipline that may help when reading a program. Perhaps more importantly, this requirement normally causes detection of the common programming error of misspelled variable identifiers.

FORTRAN and BASIC have conventions that relate the spelling of an identifier to its type, e.g, starting an integer identifier with "I" or ending a string identifier with "$". Such self-typed variables are useful in that they need not be explicitly declared and yet their type is apparent throughout the program; that is, a reader need not refer back to a declaration to determine the type. FORTRAN and PL/I have a facility whereby the program can set its own default typing conventions based on the spelling of identifiers.

2.1.1.6 Program Length -

COBOL, and to a lesser extent, Ada and PL/I, encourage a verbose style; FORTRAN programs, conversely, tend to be quite concise. The other languages, BASIC, C, and Pascal, fall somewhere in between, but probably closer to FORTRAN. It is here that we find a direct trade-off between ease of reading and writing. FORTRAN and BASIC allow the programmer to get something running with a minimum of syntactic overhead, but easily become unreadable unless the programmer is careful. Conversely, COBOL and Ada tend to be quite readable, but it takes a fair amount of effort to produce even a simple program.

2.1.2 Semantic Structure -

Semantic structure encompasses those features of the language which allow the programmer to build modules in the program to represent algorithms or data entities or both. Although the same effect can often be achieved without structures, it is vital that the language allow programs to express such structures in a natural way; this is important both for reading and writing programs.

2.1.2.1 Control Of Execution (see Figure 2) -

Control of execution addresses the language features which describe the algorithmic structure of the program. The programmer uses these features to decompose the execution sequence into logically related groups, so that the underlying design is more clearly expressed.

Figure 2 - Control of Execution (see section 2.1.2.1)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
Structured Programming	Yes	Yes	Yes	Partial/Yes	Partial	Yes	Yes
Blocks	Yes	No	Yes	No	No	Yes*	Yes
External/Internal Subroutines	Both	Both**	No	External/Both	External	Internal	Both
External/Internal Functions	Both	Both**	External	No	Both**	Internal	Both
Recursion	Yes	Yes	Yes	No	No	Yes	Yes
Generic Procedures	Yes	No	No	No	No	No	Yes
Exception Handling	Yes	Yes	No	Partial	No	No	Yes
Concurrency	Yes	Yes***	No	No	No	No	No

* No local data
** No local data for internal
*** Real-time module only

2.1.2.1.1 Structured Programming -

We use the term "structured programming" in a narrow sense to mean simply those language constructs which determine, at the detailed level, the sequence of instruction execution, and which encourage the development of well-organized source code. Such features have found their way into virtually all the languages. Except as noted below, all languages have a general purpose if-then-else, looping, and selection (case) construct. FORTRAN is the weakest in this regard; it has no single-exit select construct, and its looping mechanism depends on a control variable, not an arbitrary condition. COBOL-74's looping mechanism currently forces the performed code to be displaced out of the normal sequence, but the proposed new standard will remedy this. Other problems solved by the COBOL-8x proposal are the need for a delimiter to terminate an if-construct, but not the entire sentence (i.e., an END-IF) and the lack of a selection construct.

Ada, BASIC, and C have a statement which explicitly exits the current loop; this is useful for exiting a loop in a controlled way, rather than using the GO TO.

2.1.2.1.2 Blocks -

Blocks allow the programmer to mark off certain sections of code, such that it is treated as a single statement and, within the block, data may be defined locally which cannot be accessed from outside the block. Ada, C, and PL/I provide full block structure. Pascal's blocks do not allow the definition of local data. COBOL, FORTRAN, and BASIC do not provide this capability.

2.1.2.1.3 Subroutines -

Most languages provide a way to write subroutines which may be invoked from one part of a program and then perform some task or operate on passed parameters to return the results of the computation. External subroutines may be separately compiled, and typically communicate with their invoker through the passed parameters. Internal (or nested) subroutines are part of the same compilation unit as the invoker and typically have direct access to the invoker's data, as well as to their own local data.

C does not provide subroutines as such; invoked procedures are always external functions (see 2.1.2.1.4). Moreover, since parameter passing is always by value, the programmer must pass pointers if the function is to communicate results back to the invoker, i.e., passing parameters by reference is not provided directly, but must be simulated by the program. Pascal is weak in that it does not provide for separate compilation; all its subroutines are internal. FORTRAN has no mechanism for internal

subroutines. COBOL-74 has external subroutines, but only a weak form of internal subroutines (invoked with PERFORM) which do not accept parameters and have no local data. COBOL-8x has true internal subroutines. BASIC has internal and external subroutines, but the only local data for its internal subroutines are the parameters. Ada and PL/I have both internal and external subroutines.

2.1.2.1.4 Functions -

A function is a procedure which accepts parameters as input and returns a value. It is normally invoked as part of the evaluation of an expression. A function may be defined by the programmer or it may be supplied by the implementation (so-called intrinsic or built-in functions) as part of the standard run-time support. For user-defined functions, the distinction between external and internal described above for subroutines also applies. Only COBOL does not have the concept of functions. All the others allow the user to define such functions and require some elementary functions to be supplied by standard implementations of the language. Again, Pascal has no facility for separate compilation of such procedures.

2.1.2.1.5 Recursion -

Recursion is the ability of a procedure (subroutine or function) to invoke itself, either directly or indirectly. The definition of the procedure in the source code serves as a template and every time the procedure is invoked, the logical effect is as if a new copy of the procedure (together with its data) were created and its execution initiated. Recursion is quite valuable for several classes of algorithms. Only COBOL and FORTRAN do not have this feature.

2.1.2.1.6 Generic Procedures -

A generic procedure is a subroutine or function defined by the user which is capable of applying the same algorithm to parameters of different types. For example, a procedure which returns the largest element in an array, regardless of the type of element constituting the array, could be written, as opposed to having to write a separate procedure for each type. Only Ada and PL/I support this feature. Both allow the construction of both subroutines and functions.

2.1.2.1.7 Exception Handling -

Exception handling is the ability to have flow of control automatically transferred to a special section of code when some anomalous condition arises in the course of execution. The section of code, usually called the exception handler, may then attempt some remedial action. Languages with exception handling define a list of exceptions which implementations must detect, each identified by either a name or a numeric code. Typical exceptions are division by zero, numeric overflow, subscript out of range, faulty input, etc.

Ada, BASIC, and PL/I support exception handling. Ada and BASIC syntactically associate an exception handler with a body of code to be guarded; PL/I does this association dynamically by executing an ON statement. All three languages have a special statement which artificially causes an exception of a given type to occur. COBOL has a less general capability of declaring a procedure to be used upon the occurrence of certain I/O exceptions. Also, COBOL provides a SIZE ERROR clause on arithmetic statements for the detection of truncation, overflow, and division by zero and on the CALL, STRING, and UNSTRING statement for detection and correction of overflow.

2.1.2.1.8 Concurrency -

Concurrency (also called tasking or parallel processing) is the ability of a program explicitly to designate certain sections of code to be executed asynchronously. Logically, this means that such parallel processes should not depend on each other for their results and may be executed in any order relative to each other. Typically, there is also a mechanism which allows the programmer to synchronize these otherwise independent processes. For example, if task 3 needs intermediate results from tasks 1 and 2, task 3 can be made to wait until those results are available. Physically, concurrency may be implemented by interleaved execution on a single processor, or by true simultaneous execution on a multi-processor system. This mode of execution is in contrast to the usual "single thread" flow of control, in which, at any given time, the execution of a program has advanced to a single unambiguous point.

Only Ada, with its tasking facilities, and BASIC, with its real-time module, provide this feature. Note that COBOL provides for communication between asynchronous processes (section 2.1.4.4), but does not provide, in the language, the means of generating or controlling such processes. Although the FORTRAN standard does not support concurrency, there is a draft standard for Industrial Real-Time FORTRAN [IRTF84], developed by the European Workshop on Industrial Computer Systems Technical Committee 1 (EWICS/TC1) and reviewed by Working Group 1, Programming Language for the control of industrial processes (ISO/TC97/SC5/WG1), which is currently in the approval process

within ISO/TC97/SC5 (see 2.1.4.3, below). This draft specifies standard library routines, through which FORTRAN can perform parallel processing.

2.1.2.2 Control Of Data (see Figure 3) -

This section discusses the mechanisms available to the programmer for establishing the logical appearance of data within the program. The concern here is not with actual data values and operations, which are covered in section 2.1.3. Rather this section deals with the extrinsic characteristics of data, such as its scope, lifetime, and other logical properties.

2.1.2.2.1 Storage Classes -

Data can be established, stored, and deleted in various ways. There are both logical and physical implications of each technique. Usually, data is associated with the particular program-unit (function or subroutine) in which it is defined and all languages have rules about the treatment of such "local" data. Typically, data declared in a procedure may be either static or automatic. When data is static, its value is retained between invocations of the procedure, and normally storage for it is allocated only once before execution of the program. In the case of automatic data, each invocation generates a new instance of the data, and the usual implementation technique is for storage to be allocated and de-allocated for each entry to and exit from the procedure.

The default for all the languages is automatic, except for COBOL, in which the default is static. C, FORTRAN, and PL/I support static data, but Ada, BASIC and Pascal do not. COBOL has an executable statement, CANCEL, which causes a fresh copy of a designated subroutine to be used at the next invocation. COBOL-8x has a declarative phrase, INITIAL, which specifies that the program-unit containing it is to be initialized upon each entry. Both CANCEL and INITIAL give the effect of automatic.

The languages with pointers (see 2.1.3.2.5, below), Ada, C, Pascal, and PL/I, all allow the program explicitly to create and destroy multiple instances of data to be addressed by the pointer.

PL/I also has a storage class, CONTROLLED, in which the program explicitly creates and destroys instances of a data-type according to a stack discipline (last-in, first-out). C has a storage class REGISTER, which is logically similar to automatic, but tells the compiler to attempt to maintain the data item in a register for fast access.

Figure 3 - Control of Data (see section 2.1.2.2)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
Storage Classes:							
automatic	Yes*	Yes*	Yes*	Yes	Yes*	Yes*	Yes*
static	No	No	Yes	Yes*	Yes	No	Yes
user-allocated	w/pointer	No	w/pointer	No	No	w/pointer	w/pointer, stacked
register	No	No	Yes	No	No	No	No
External Data	Yes	Yes**	Yes	No/Yes***	Yes***	N/A	Yes***
Data Abstraction:							
User-named types	Yes	No	Yes	No	No	Yes	No
Type-checking	Yes	N/A	No	N/A	N/A	Yes	N/A
User-defined operations	Yes	N/A	No	N/A	N/A	Partial	N/A

* Default
** Real-time module only
*** Must be re-declared

2.1.2.2.2 External Data -

A useful feature is the ability explicitly to declare some data as external (also called common) and therefore accessible to the entire program, including external subroutines and functions. Ada, C, COBOL-8x, FORTRAN, and PL/I all have this capability. COBOL-8x, FORTRAN, and PL/I, however, require that external data be declared in every procedure which accesses it. Thus, it is external in the sense that there is one copy of the data available at run-time, but not in the sense that one copy of the source code declarations is visible throughout the program, i.e. the external data promotes run-time efficiency, but not ease of source code development. COBOL-74 does not support external data and BASIC allows it only in its real-time module, not in the core of the language. Since Pascal doesn't have external procedures, it obviously has no mechanism for setting up data accessible to such procedures.

2.1.2.2.3 Data Abstraction -

Data abstraction is a feature which is strongly characteristic of the newer languages. Essentially, it is the ability to describe data according to its logical (abstract) meaning, rather than according to the way it is to be represented internally. This is important both for scalar and aggregate data. Taking a simple example, to define a data item as FLOAT(6) merely stipulates that the item will contain real numbers of a certain precision as values. To define the same item as VELOCITY (where VELOCITY is elsewhere defined as FLOAT(6)), preserves more information about the intended use of the item.

We can distinguish several levels of capability to define data abstractly. The most basic is the ability to give a name to a particular data type and thereafter use that name when defining data items. This has two advantages: first, data items are described in terms of their logical meaning, rather than their physical implementation (VELOCITY as opposed to FLOAT(6)), and secondly, if that implementation needs to be changed, it can be done at one place in the program (e.g., by changing the definition of VELOCITY to FLOAT(8)), and the change then automatically propagates throughout the program. Ada, C (with its TYPEDEF clause), and Pascal all have this capability.

The next level of capability is for the language to recognize user-defined types and automatically provide type-checking (see 2.1.3.1, below) and some operations, such as assignment and comparison, for them. Ada and Pascal provide these features.

Finally, the user may wish to define new operations on the data type. Ada provides many features to support this ability. Pascal supports it only through the ability to define functions and subroutines, using the new data type.

2.1.2.3 Packages -

If we view semantic structure as a way of expressing a program according to its logical behavior, rather than its physical implementation, then packages represent the highest level of abstraction. In packages, there is no need to declare operations and data abstraction separately. Rather, the program can define logically related data, data-types, and operations (as specified in associated subroutines and functions) together in a package. The package provides a logical environment which can then be invoked by any program, or part of a program, needing those capabilities. Packages could be built to support such facilities as indexed files, relational databases, graph manipulation, rational arithmetic, and so on. Only Ada provides packages.

2.1.3 Data Types And Manipulation -

Perhaps the most salient feature serving to distinguish the various languages is the choice of data types each offers and the operations available on that data. While the older languages concentrated on providing data types which would be appropriate for their application domain, the newer languages, Ada in particular, have provided mechanisms for the user to define his own data types. This strategy has the advantages of user flexibility and language economy, but the penalty of requiring extra work for the user, and a potential degradation of portability (e.g., if each user defines fixed-point decimal differently). This section will consider only data types and operations supplied directly by the standard implementations of the language.

2.1.3.1 Checking And Coercion -

Strict type checking prevents certain classes of error from being committed by programmers, in that the language allows only some carefully restricted combinations of data types and operations. Thus, type-checking emphasizes discipline and control, and the avoidance of surprising results. The disadvantage of this approach is that it is sometimes awkward to perform operations which intuitively seem simple. Conversely, type coercion allows a wide variety of interaction among data types (automatic conversion), giving the user more power and flexibility, but is more subject to misuse.

Ada and Pascal take a rather strict approach, allowing only a very few coercions. PL/I is at the other extreme; its specification includes very complex semantics dealing with conversion rules. The other languages fall somewhere in-between, allowing certain conversions, but forbidding those which are plainly mistaken.

2.1.3.2 Elementary Data -

Elementary data-types (also called scalar) are those which represent a single indivisible value, as opposed to aggregate types (covered below) which represent some combination of values. Normally, the language standard simply describes the logical properties associated with data defined as a certain type, but not its underlying representation. In COBOL and PL/I, however, the programmer may describe data with so-called "pictures", which imply a character-oriented representation of the data.

2.1.3.2.1 Numeric (see Figure 4) -

All the languages can represent numeric data in some form. PL/I is the most inclusive, covering all the types specified in the other languages. All the languages have some form of floating-point numbers, except COBOL. Most also provide a second floating-point type with extra precision, but Pascal does not. BASIC is notable in that its longer floating-point type is defined in terms of decimal. Thus, the user may choose between greater accuracy and predictability on the one hand, and execution efficiency on the other. Fixed-point numbers are available in PL/I, Ada, BASIC, and of course COBOL. Ada's fixed-point numbers are, however, defined as binary, rather than decimal. An integer type which permits direct binary hardware implementation is available in all the languages except COBOL-74 and BASIC. Of course, with fixed-point data the user can achieve the logical effect of integer data. Complex numbers are provided only in FORTRAN and PL/I.

All the languages provide the four elementary arithmetic operations (addition, subtraction, multiplication, and division), and comparisons (less-than, etc.) for numeric data. C and COBOL are unique in providing special syntax for incrementing and decrementing a variable, in addition to the general capability of assigning the value of an arbitrary expression to a variable. Pascal and C do not have exponentiation, and Ada provides exponentiation only to an integer power. Beyond this, there is a wide variety of functions provided for numeric data. BASIC, FORTRAN, and PL/I have a large set of supplied functions, including many transcendental functions (hyperbolic, trigonometric and logarithmic). Pascal has a somewhat smaller set, but still including some trigonometrics and logarithmics. Ada provides only absolute value and remainder functions. C and COBOL provide none.

Figure 4 - Numeric Data and Manipulation (see section 2.1.3.2.1)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
Long/Short Floating-point	Both	Both*	Both	No	Both	Short	Both
Binary/Decimal Fixed-point	Binary	Decimal	No	Decimal	No	No	Both
Integer	Yes	No	Yes	No/Yes	Yes	Yes	Yes
Complex	No	No	No	No	Yes	No	Yes
Add, Subtract, Multiply, Divide	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Exponentiation	Yes**	Yes	No	Yes	Yes	No	Yes
Numeric Functions	Few	Many	None	None	Many	Some	Many

* Long format is decimal
** Raising to integer power only

2.1.3.2.2 Character (see Figure 5) -

All the languages provide a way of storing characters. Ada, Pascal, and C use arrays to handle sequences of characters, whereas the other languages treat the string as a special type of aggregate. BASIC, C, and PL/I provide direct support for variable-length strings (with a declared or default maximum length); in the other languages, character strings are inherently fixed-length. Pascal has very limited character manipulation; only assignment and comparison are provided.

All the languages except Pascal support the concatenation of several character items into one variable. All except C, COBOL-74, and Pascal allow specification of a substring by position (e.g., select the 2nd through 5th character in a string). The proposed COBOL-8x standard provides full substring capability. COBOL and PL/I can retrieve (but not assign to) a substring based on some delimiter in the string itself (e.g., select everything preceding the first comma).

BASIC, COBOL, FORTRAN, and PL/I all have a means of searching for the occurrence of a character string within another string. C, COBOL, and PL/I provide a way to test whether characters are alphabetic, all upper- or lower-case, or numeric. COBOL-8x and PL/I provide a facility to translate systematically from one set of character values to another (e.g., translate all A's to X, all B's to Y, all C's to Z).

All the languages (besides Pascal) provide one or more mechanisms to convert between other data-types in the language and equivalent character strings; COBOL's conversion from string to number, however, requires prior specification of the format to be converted from. Ada, C, FORTRAN, and PL/I achieve such conversions by pseudo-IO statements, which, instead of accessing a true external file, access a string variable. Ada, BASIC, and PL/I provide conversion functions. COBOL and PL/I do many conversions as a result of coercing one data-type to another upon assignment.

2.1.3.2.3 Logical (see Figure 6) -

Logical data is capable of taking on only two values, "true" and "false", and serves as a condition which can be tested in various control statements. Most of the languages have such a type (also called boolean), but support it in different ways. Ada, Pascal, and FORTRAN support this type directly. BASIC does not have logical data. COBOL has named conditions, which can simulate the effect of logical data. C uses the numeric values of zero and non-zero to stand for false and true. Finally, PL/I has a BIT type (see below), which has only two values, 0 and 1, and which can be used as a direct simulation of logical data.

Figure 5 - Character Data and Manipulation (see section 2.1.3.2.2)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
Aggregate	Array	String	Array	String	String	Array	String
Fixed/Variable Length	Fixed	Variable	Variable*	Fixed	Fixed	Fixed	Both
Concatenation	Yes	Yes	Yes	Yes	Yes	No	Yes
Substring by position:	Yes	Yes	No	No/Yes	Yes	No	Yes
by contents:	No	No	No	Yes	No	No	Yes
String Search	No	Yes	No	Yes	Yes	No	Yes
Test Upper/Lower Case, Numeric	No	No	Yes**	Yes	No	No	Yes
Translate	No	No	No	No/Yes	No	No	Yes
Conversion Mechanism	Functions Pseudo-IO	Functions	Pseudo-IO	Co-ercion	Pseudo-IO	None	Functions, Pseudo-IO, Co-ercion

* within fixed-length array
** individual characters only

Figure 6 - Logical Data and Manipulation (see section 2.1.3.2.3)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Logical Data Implementation	Logical	None	Numeric Value	Named Condition	Logical	Logical	Bit
Operations:							
and, or, not	Yes	N/A*	Yes	No*	Yes	Yes	Yes
exclusive-or	Yes	N/A	No	No	Yes	No	No
equivalence	No	N/A	No	No	Yes	No	No

* and, or, not available for conditions, but not data

Except for BASIC and COBOL, the languages allow storing the result of a logical expression formed from logical data and operators. All the languages allow combining conditions (as opposed to data) with the logical operators and, or, and not (conjunction, disjunction, and negation). Ada also has the "exclusive or". FORTRAN has the exclusive or and logical equivalence. PL/I, in addition to specific syntax for conjunction, disjunction, and negation, has a general purpose BOOL function which may be used to obtain any of the 16 possible boolean binary functions.

2.1.3.2.4 Bit (see Figure 7) -

Bit data is capable of taking on only two values, 0 and 1. Normally, bit data is implemented directly on the hardware bit structure of the underlying machine.

Only C and PL/I support bit data. As mentioned above, PL/I also uses the bit type as its logical type. Both C and PL/I allow bitwise logical operations on bit aggregates (i.e., the operation is performed on pairs of bits, one from each aggregate in corresponding positions), such as logical and, or, and not. C also supports the exclusive-or operation and shifting. C uses one (integer) machine word as the unit for bit-aggregation. PL/I aggregates bits into strings of arbitrary length; the operations available for character strings (concatenation, substring, search) also apply to bit strings. Also, PL/I provides two functions, SOME and EVERY, which test whether some bit or every bit in a string has a value of 1. The FORTRAN standard [FORT78] does not support bit data, but there is a draft standard for Industrial Real-Time FORTRAN [IRTF84] which does provide bitwise operations on integers, much like C (see section 2.1.4.3, below).

2.1.3.2.5 Pointer (see Figure 8) -

Pointer data is that which can be used to address, or point to, other data objects. It is useful in constructing various kinds of dynamic data structures, such as lists, stacks, trees, graphs, and queues. The ability to allocate and free storage is typically associated with this data type (see 2.1.2.2.1). Ada, Pascal, C, and PL/I have these facilities; BASIC, COBOL, and FORTRAN do not.

2.1.3.3 Aggregate Data -

Aggregation is any mechanism in the language for building up a data structure out of smaller pieces, such as smaller aggregates or elementary data, together with the operations provided to manipulate these structures.

Figure 7 - Bit Data and Manipulation (see section 2.1.3.2.4)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Bit Data	No	No	Yes	No	No	No	Yes
Aggregate	N/A	N/A	Machine Word	N/A	N/A	N/A	String
Bitwise: and, or, not	N/A	N/A	Yes	N/A	N/A	N/A	Yes
exclusive-or	N/A	N/A	Yes	N/A	N/A	N/A	No
Shift	N/A	N/A	Yes	N/A	N/A	N/A	No
Substring, Search, Concatenation	N/A	N/A	No	N/A	N/A	N/A	Yes
Test Some, Every Bit	N/A	N/A	No	N/A	N/A	N/A	Yes

Figure 8 - Pointer Data and Manipulation (see section 2.1.3.2.5)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Pointers and Allocation	Yes	No	Yes	No	No	Yes	Yes

2.1.3.3.1 Arrays (see Figure 9) -

Arrays are ordered collections of data of similar type. Each element of an array can be addressed individually according to some identifying scalar value (usually integer values, but sometimes other types as well). Arrays can be nested so as to provide the effect of multidimensional entities.

All the languages support arrays in one way or another. Only Ada and Pascal allow addressing by other than integer values; addressing may also be done with so-called enumeration types (any type with a set of discrete, ordered values). BASIC and COBOL-74 set a rather low limit of three on the number of dimensions supported. COBOL-8x will allow at least seven. C and COBOL fix the lower bound subscript of arrays to 0 and 1, respectively; the other languages allow the user to specify a lower bound. Ada, BASIC, Pascal, PL/I, and COBOL allow simple array assignment. Comparison of two arrays (at least for equality) is allowed only by Ada, COBOL, and PL/I. COBOL, however, performs these operations by character, not by element. C and FORTRAN provide no array operations.

Because of the potentially large number of values involved, it is useful to be able to initialize arrays other than by individual assignment to each element. Ada, C, COBOL, FORTRAN, and PL/I provide a way of initializing arrays with a list of values in the declaration of the array, although the COBOL mechanism is less convenient than the others. C allows such initialization only for external or static arrays, not for automatic arrays (see section 2.1.2.2.1). Ada, COBOL-8x, FORTRAN, and PL/I all have a simple way to set all elements of the array to a single value in the array declararion. The Ada mechanism for initialization can also be used in executable statements. Similarly, BASIC's DATA statement can be used for assigning a list of values to an array (albeit within a loop), and COBOL-8x and PL/I can set all the elements to a single value during execution.

BASIC and PL/I allow arithmetic and string operations on arrays as a whole, as well as by individual element. For example, the program may multiply the elements of one array to those of another and store the result in a third array with a single statement. PL/I always does such operations element-by-element, but BASIC treats numeric arrays according to the rules for matrix arithmetic.

COBOL is unique in providing a statement which automatically searches an array for a given value. If the array is ordered, a binary, as opposed to linear, search can be performed.

Figure 9 - Arrays (see section 2.1.3.3.1)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
Dimensions	no limit	3	no limit	3 / 7	7	no limit	no limit
Subscript Type	Discrete	Integer	Integer	Integer	Integer	Discrete	Integer
Set Lower Bound	Yes	Yes	No	No	Yes	Yes	Yes
Array Operations:							
assignment	Yes	Yes	No	Yes*	No	Yes	Yes
comparison	Yes	No	No	Yes*	No	No	Yes
initialization:							
compile-time							
one value	Yes	No	No	No/Yes	Yes	No	Yes
value list	Yes	No	Yes	Yes	Yes	No	Yes
run-time							
one value	Yes	No	No	No/Yes	No	No	Yes
value list	Yes	Yes	No	No	No	No	No
arithmetic	No	Yes**	No	No	No	No	Yes
string	No	Yes	No	No	No	No	Yes
search	No	No	No	Yes	No	No	No

* as character string, not element-by-element
** as matrices, not element-by-element

2.1.3.3.2 Files And I/O (see Figure 10) -

Files are organized collections of data which may exist outside the program. Operations associated with files are reading, writing, deleting and modifying elements of the file. The latter two operations are often not available for sequential files.

Sequential files are those in which the data exists in linear order and can be accessed only with respect to the current position in the file and to the beginning and end of the sequence. All the languages support sequential files. COBOL is unique in providing a facility to sort a file ordered by the contents of any selected fields, generating a new sequential file. Also, in COBOL, a group of files ordered by the same fields may be merged to form one ordered file.

Relative files (also called "direct") are those in which each position in the file is numbered sequentially, and may be accessed by means of the identifying number. Only C and Pascal do not support such files.

Finally, keyed files are those in which each element of the file has an associated character string identifier, called its key. The normal operations for keyed files are the ability to access individual elements by key, and also to access them sequentially in key order. BASIC, COBOL, and PL/I have keyed files.

A file may be broadly classified as internal or external, according to the nature of its constituent elements. Elements of internal files typically are some type of data aggregate, often records, with logically related components organized in one or a few well-defined formats. They are usually encoded for efficient I/O operations and used for long-term storage of data.

All the languages except C provide internal files. COBOL and PL/I use records as the elements of the file. Ada and Pascal support files composed of any defined data-type, including records. BASIC and FORTRAN simply operate with lists of variables or expressions which are implicitly treated as a unit for file operations. BASIC also has a TEMPLATE which can be used to impose a fixed format on the list.

External files are display oriented. Their elements are usually strings of characters, called lines, in either fixed or free format. Fixed format implies that the program explicitly specifies the representation of data within the line to be input or output. Free format implies that the character representation of values is done automatically on output, and scanned and parsed automatically on input. Typically, external files handle only elementary data, not aggregates. They are useful for human interaction and for data interchange between systems with different internal representations of non-character data, such as floating-point numbers.

Figure 10 - Files and I/O (see section 2.1.3.3.2)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
File Organization:							
Sequential	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Relative	Yes	Yes	No	Yes	Yes	No	Yes
Keyed	No	Yes	No	Yes	No	No	Yes
Sort/Merge	No	No	No	Yes	No	No	No
Internal File Aggregate	Any Type	List (TEMPLATE)	None	Record	List	Any Type	Record
External Files:							
Fixed Format I/O	I/O	Output	I/O	I/O	I/O	Output	I/O
Free Format I/O	I/O	I/O	I/O	Output*	I/O	I/O**	I/O
Line I/O	I/O	I/O	I/O	I/O	Output	I/O	Output
Partial Line I/O	I/O	Output	I/O	No/Output	No	I/O	I/O
Low Level I/O	I/O	No	I/O	No	No	I/O	No

* Only for certain hardware devices
** Input works only with numbers

Ada, C, FORTRAN, and PL/I all support external files which may have a fixed or free format both on input or output. BASIC and Pascal do not have a mechanism for fixed format on input. COBOL does not provide free format on input (not counting the ACCEPT verb, which inputs only a single character string), a serious impediment to interactive applications. The only free format output it provides (with the DISPLAY verb) may not be available for all files. Pascal provides free format input only for numbers, not for booleans or character strings. All the languages except FORTRAN and PL/I provide a way to read an entire line (whose length is not known in advance) from a terminal as a single character string. Ada, C, Pascal, and PL/I are all capable of reading or writing part of a line as a single operation. COBOL-74 and FORTRAN do not handle partial lines. BASIC and COBOL-8x can write, but not read, a partial line.

Finally, some languages have a way to do low-level I/O, in which very small pieces of data, such as individual characters, can be manipulated. Ada provides a syntactic framework for low-level I/O which is fully specified in a device-dependent way. Pascal and C can both access files one character at a time.

2.1.3.3.3 Records (see Figure 11) -

Records (also called "structures") are a way of grouping together logically related data of different types. Very often, file I/O is performed at the record level. Internal operations normally associated with records are assignment, comparison, and parameter-passing.

Only BASIC and FORTRAN do not have a record type such as to allow internal operations. The operations allowed on records in C are quite limited; in particular, I/O cannot be done at the record level, nor are any of the internal operations available. C manipulates records almost solely through the use of pointers to those records. The pointers can, of course, be assigned, compared, and passed as parameters. Ada, COBOL, Pascal, and PL/I all allow record assignment and parameter passing. In addition, COBOL and PL/I allow assignment between elements of different types of records, based on the correspondence of names of those elements. Pascal does not have record comparison; Ada and PL/I allow comparing only for equality or inequality; COBOL allows ordered comparisons (less-than, etc.) as well, treating records as character strings.

Most of the languages support variable-format record definitions (possibly of different lengths) for the same area of storage; this is needed for handling files with formatted records, but more than one such format. Ada and Pascal provide a record definition wherein the contents of a particular field (often called the record tag) determines which of several formats applies. C, COBOL, FORTRAN and PL/I simply allow re-defining the same area, leaving control to the programmer. FORTRAN also has

Figure 11 - Records (see section 2.1.3.3.3)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Internal Records	Yes	No	Yes	Yes	No	Yes	Yes
Record Operations: assignment:							
entire record	Yes	N/A	No	Yes	N/A	Yes	Yes
corresponding name	No	N/A	No	Yes	N/A	No	Yes
comparison	Yes	N/A	No	Yes	N/A	No	Yes
Variable Format Mechanism	Defined Tag	None*	Redefined Area	Redefined Area	Redefined Area	Defined Tag	Redefined Area

* Re-read record to handle different formats in file

"left-tabbing" to permit re-interpretation of the same input field. BASIC handles the problem not through a record definition, but by providing for reading only part of a record and then re-reading it.

2.1.3.3.4 Sets (see Figure 12) -

A set is an unordered collection of elements chosen from a specified universe. The usual operations associated with sets are testing whether an element belongs to a set, inserting or deleting an element, testing whether one set is a subset of another, and taking the union, intersection, difference, or negation of sets. Also, it is possible to express a literal value to be assigned or compared to a set variable with a set constructor.

Only Pascal directly supports the concept of a set. BASIC, COBOL, and FORTRAN do not support this type at all. The other languages have features which permit a rather direct simulation and so we shall discuss them here even though their facilities are defined at a less abstract level. Bit strings can be used in PL/I (to simulate sets of integers only), arrays of booleans in the case of Ada, and C has bitwise operations on integers along with the ability to define bit-fields. In the case of C and Pascal, the size of the universal set can be quite small, often limited to the word size of the machine. Although not all operations are directly supported by these four languages, it is usually straightforward to express a missing operation in terms of those provided, e.g., the set difference, A - B, can be computed as A AND (NOT B). Ada, C, and PL/I all directly support unary negation, but not set difference, nor subset testing. Conversely, Pascal supports difference and subset testing, but not negation.

2.1.4 Application Facilities (see Figure 13) -

Application facilities characterize the major semantic functions of a language beyond those associated with particular data-types. Also, these facilities tend to be closely related to the external entities upon which the program operates and thus visible to the end-user, as opposed to the internal features of the language, visible only to the programmer.

2.1.4.1 Reports -

Only COBOL, with its report writer facility, directly supports the generation of reports. The essential feature of report writer is that the report's format is declared statically, rather than generated as the result of an explicit algorithm.

Figure 12 - Sets (see section 2.1.3.3.4)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Set Type Implementation	Boolean Array	None	Bit Data, Fields	None	None	Set	Bit Data
Set Operations: Union	Yes	N/A	Yes	N/A	N/A	Yes	Yes
Intersection	Yes	N/A	Yes	N/A	N/A	Yes	Yes
Difference	No	N/A	No	N/A	N/A	Yes	No
Negation	Yes	N/A	Yes	N/A	N/A	No	Yes
Subset Test	No	N/A	No	N/A	N/A	Yes	No
Equality Test	Yes	N/A	Yes	N/A	N/A	Yes	Yes
Constructor	Yes	N/A	Yes	N/A	N/A	Yes	Yes
Element Operations: Belonging Test	Yes	N/A	Yes	N/A	N/A	Yes	Yes
Add, Delete	Yes	N/A	Yes	N/A	N/A	Yes*	Yes

* By using single-element set literal

Figure 13 - Application Facilities (see section 2.1.4)

LANGUAGE FEATURES	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Reports	No	No	No	Yes	No	No	No
Data Base	No	No	No	No*	No	No	No
Real-time	Yes	Yes	No	No	No*	No	No
Communication	Yes	Yes	No	Yes	No	No	No
Graphics	No*	Yes	No*	No	No*	No*	No

* standardization effort in progress

2.1.4.2 Database -

While many language interfaces to database facilities are commercially available (especially for COBOL and FORTRAN), no standard currently exists for such interfaces. Standardization work is furthest advanced for COBOL. ANS committee X3H2 has primary responsibility for database standards within the U.S. It may be expected that when X3H2 completes its work on a database standard, bindings to the various languages will follow shortly [Gall84].

2.1.4.3 Real-time -

Real-time applications are those in which the program is controlling some activity for which time is an important constraint, not simply in the sense of completion of the task, but that the progress of the activity is tied to true physical time. Typical examples are the control of machinery in a factory or of devices in vehicles such as aircraft. This is in contrast to such applications as file processing, in which only the logical behavior is important and there is no strong dependence on physical time.

To support such applications, a language must have a way of expressing relationships to real time (either absolute time or duration) and handling asynchronous events, such as interrupts. Also, such applications usually involve concurrency (see 2.1.2.1.8, above). Only Ada, with its tasking facility, and BASIC, in its real-time module, support real-time applications. As mentioned above in the section on concurrency, there is a draft standard for Industrial Real-Time FORTRAN [IRTF84], which provides real-time support.

2.1.4.4 Communication -

Communication is the ability for one process to pass information to another process which may be executing asynchronously. One process is a sender and the other a receiver of a message. The actual transmission of the message may occur at the same time for both processes, or the system may accept a message from a sender and transmit it later to a receiver. Only Ada, BASIC (in its real-time module), and COBOL provide this facility. Ada and BASIC use synchronized messages (send and receive at the same time), while COBOL allows queueing of messages.

2.1.4.5 Graphics -

Graphics is the capability of manipulating visual objects and properties, such as locations, lines, two- or three-dimensional shapes, and colors. Typically, the main interest is in generating displays on a graphics screen, and accepting information from such a screen in terms of an indicated location or area. ANS committee X3H3 has primary responsibility for graphics within the US and they work closely with the international standardization efforts of ISO/TC97/SC5/WG2. The BASIC standard incorporates a set of facilities from the ISO draft standard for graphics. Work is under way for bindings of most of the other languages to the draft standard, with FORTRAN being the farthest advanced.

2.1.5 Program Implementation Control -

Some of the languages have features which enable the programmer to control the compilation and execution process, so as to tailor a program to a given machine, or to some optimization goal. These features differ from those described earlier in section 2.1 in that their primary goal is not to define the logical behavior of the program; nonetheless, that behavior is sometimes affected. The use of these features is not without cost. By their nature, they usually involve non-portable code. Some of their functions could be performed using system control commands, thus limiting the program to a purely logical description of the application. On the other hand, they do provide a somewhat vendor-independent way of specifying common implementation options.

Ada has a quite extensive set of facilities to control the way programs are implemented. The two mechanisms provided are so-called pragmas and representation clauses. Pragmas are available to tell the compiler about the memory size of the machine, to optimize storage space or time in general, and to optimize storage for chosen data items, among other functions. Representation clauses tell the compiler how to map certain logical entities in the program to the underlying machine architecture.

BASIC has an OPTION statement, some clauses of which are logical in nature, but two have efficiency implications. The program can specify the use of the standard ASCII collating sequence for character comparison, or of the possibly different native sequence. Also, the ARITHMETIC option determines whether numeric operations will be performed according to a decimal model defined in the standard, or an undefined native model.

COBOL's CONFIGURATION SECTION has clauses to tell the compiler about available memory and to define the character collating sequence. The segmentation feature tells the system how to overlap storage when executing the program. The USAGE and

SYNCHRONIZATION clauses determine the hardware representation of data items. Certain physical properties of files and buffer areas may be specified in the I-O-CONTROL paragraph and FILE SECTION.

Pascal's only feature for program implementation control is the packed/unpacked attribute for data (packed implies using less storage than unpacked). There is a compiler directive, "forward", which is not for program implementation control (it is a syntactic device to declare the existence of a procedure farther on in the program), but implementors may define their own directives for such purposes, much like Ada's pragmas.

PL/I provides an OPTIONS statement, but the standard does not define any particular options - this is left to implementors. For describing files, PL/I's ENVIRONMENT attribute allows the specification of implementation-dependent characteristics. Data items may be specified as ALIGNED or UNALIGNED, directing that data be either aligned on a storage boundary (usually implying faster access), or packed together regardless of boundaries (and therefore using less space).

As mentioned in section 2.1.2.2.1, C has a storage class REGISTER, which informs the compiler to keep the data item stored in a register for fast access, if possible.

FORTRAN has no facilities for program implementation control.

2.1.6 Simplicity -

Language simplicity is difficult to measure quantitatively, although there have been some attempts [Hals77]. We shall rely on the intuitive notion that a language with relatively few semantic concepts and syntax rules is simpler than one which embodies several more sophisticated concepts, and more complex syntactic expressions. Simplicity is not good, per se - complex applications will likely require a complex language. Simplicity, however, makes for easier learning and fewer errors in actual use. Certainly, the goal is that a language be as simple as possible, given the semantic domain it must cover.

BASIC is the simplest language under study. There are only two data types available within a given program, variable-length strings, and numbers. It is possible to perform I/O operations with simple defaults that require no formatting. The convention of one statement per line avoids some pitfalls that may arise with statement separators and terminators.

FORTRAN and C also allow the writing of comparatively short, straightforward programs, although there is a bit more syntactic overhead than with BASIC.

Pascal is somewhat more complex. Its structure is very regular, which contributes to simplicity, but it requires relatively elaborate declarations, and its rules governing statement groupings and separators can lead to subtle program errors.

Finally, Ada, COBOL, and PL/I must be rated as the most complex of the languages. This is understandable, given the applications for which they were designed. Nonetheless, it requires substantially greater effort to master all the rules governing the use of one of these languages.

2.1.7 Standardization (see Figure 14) -

The advantages of standardization have already been covered in the preface and need not be repeated here. Factors which support language standardization include the existence of a formally sanctioned specification, present and future availability of conforming implementations, and a mechanism for conformance testing of language processors. Also relevant to standardization is the availability of software tools to monitor the conformance of programs to a language standard - see section 2.1.10.1, below.

Of the languages covered in this report, COBOL and FORTRAN are the most thoroughly standardized. They have long been the subject of ANSI standards activities, and conforming implementations are widespread throughout the industry. Furthermore, they have been adopted as FIPS, and implementations are subject to validation by the General Service Administration's Federal Software Testing Center (GSA/FSTC). Although there are ongoing efforts to revise these ANSI standards, it is likely that future versions will be substantially compatible with the current specifications. Note especially that the revision of COBOL is in the later stages of the approval process within ANSI and ISO [COBO83]. ICST is actively considering the adoption of this new version of COBOL as a FIPS.

Pascal and Ada are both recently adopted (1983) ANSI standards and so there are fewer conforming implementations than for COBOL and FORTRAN. It is highly likely that conforming implementations will become widely available. Fortunately, neither language has been subject to great differences in implementation, in the case of Pascal because the original Pascal report [Jens74] served as a de facto standard, and in the case of Ada, because the standard preceded actual implementations. At the time of this report, ICST is actively considering the adoption of Ada and Pascal as FIPS. Validation tests are available for both languages. In the case of Ada, these are formally administered by the Ada Validation Organization.

Figure 14 - Standardization (see section 2.1.7)

TYPE OF STANDARD	Ada	Proposed BASIC	C	COBOL-74/8x	FORTRAN	Pascal	PL/I
FIPS	Planned	Planned	To be decided	21-1*/Planned (1975)	69* (1980)	Planned	No
ANSI	MIL-STD-1815A (1983)	Planned	Planned	X3.23/Planned (1974)	X3.9 (1978)	IEEE770 X3.97 (1983)	X3.53 (1976)
ISO	Planned	Planned	To be decided	1989/Planned (1978)	1539 (1980)	7185** (1983)	6160 (1979)

* adopts ANSI standard
** differs from ANSI standard

The British Standards Institution (BSI) offers a Pascal Compiler Validation Service. These tests however, are based on the ISO standard for Pascal [ISO83], not the ANSI standard. The ISO standard has two levels, level 0 being substantially the same as the ANSI standard, and level 1 including an additional feature (conformant arrays) which allows an invoked procedure to accept an array of any size as a parameter.

There has been an ANSI standard for PL/I since 1976. Although those implementations developed since that time have generally followed the standard, there have been relatively few such implementations. No comprehensive validation tests are available. In 1981, an ANSI standard was adopted for a subset of PL/I, and this appears to be gaining more acceptance, not only among mainframes and minis, but also for micros. Nonetheless, it must be said that standardization for PL/I has not been as effective as for the languages discussed above. The future prospects, especially for the full language, are uncertain.

There is an ANSI standard, which has been adopted as a FIPS, for Minimal BASIC, a very small subset of the language considered throughout this report. FSTC is performing validations for the Minimal BASIC standard. Although many implementations conform to this standard, the language it describes is so small that there are many implementor-defined enhancements. Furthermore, unlike Pascal and Ada, in the absence of a de facto standard, large differences have evolved in BASIC implementations. The proposed standard for a complete version of the language should help remedy this situation, but full standardization is coming to BASIC much later than to COBOL and FORTRAN. As a result it will be a few years before the support for the BASIC standard can be as comprehensive as for other languages. The experience with Minimal BASIC gives grounds for optimism, but the acceptance of full BASIC as a standard is not as assured as for Ada or Pascal.

Finally, development of an ANSI standard for C has begun only recently. Like Pascal, the existence of a de facto standard [Kern78] has helped forestall wide divergence in implementation, but there are some differences, especially in the runtime library. Indeed, one of the outstanding questions is the degree to which a C standard will be divorced from UNIX runtime support. No comprehensive validation test set yet exists. It is too early in the process to be able to estimate the acceptance of a C standard.

2.1.8 Performance -

Strictly speaking, performance is a characteristic of a given implementation of a language, not of a language as such. Nonetheless, the nature of a language often encourages a certain type of implementation strategy. For instance, language design usually presents the problem of trading off a large number of features against compiler size and speed. As a language

implementation consumes more machine resources, it becomes more expensive to implement a given application. Resources include both storage and time and may be consumed during various phases of processing, such as interpretation, compilation, and execution. The relationship between language and performance may be complex. For instance, a higher level language feature such as assignment of entire arrays may place a greater burden on the compiler, yet also facilitate run-time optimization. Of course the relative importance of compiler efficiency vs. execution efficiency depends on the application.

Generally speaking, C, FORTRAN, and Pascal require relatively few resources. These languages were designed for both easy compilation and rapid execution.

BASIC is often implemented with an interpreter, which of course implies diminished overhead (no compilation), but slow execution. BASIC compilers are available, however, and such implementations may be expected to be roughly comparable to FORTRAN in performance.

The design goals for Ada are just the opposite as for an interpreter. Given the elaborate mechanisms for declarations and for linking packages together, compilation is likely to be rather expensive. Ada programs are supposed to operate in a real-time environment, however, and execution should be fairly efficient, although some run-time detection of exceptions may be expensive. Judgments about Ada's performance, however, must be tentative, pending the accumulation of experience with actual implementations.

Finally, COBOL and PL/I, as the largest languages under study, have a relatively high cost for compilation and execution, especially compilation. There are implementations of lower levels of COBOL, and of the PL/I subset; these implementations have an improved level of performance, as suggested by the fact that they run on mini- and microcomputers.

2.1.9 Software Availability -

Sometimes, the easiest way to develop a program is to buy one that has already been written. To the extent that re-usable and supported code is available in one language rather than another, it becomes advantageous to use that language, especially if the purchased software must interact with user-written programs.

Ada, as a new language, does not have a significant software base. The design goals of the language, however, place great stress on re-usability of code, as epitomized in Ada's "package" feature. It is reasonable to expect that, as the language comes into use, a good deal of software will be available.

Almost all systems programming for UNIX is done in C, and hence there are many system routines available in that language which may be accessed through the library mechanism, when C is running under UNIX.

There are many application packages (see Appendix C) available in COBOL, covering most common data processing functions. It is often possible to buy a complete application outright and then simply tailor and maintain the code.

FORTRAN, as the dominant language for scientific and engineering applications, has extensive packages and libraries in those areas. Especially in the field of mathematical software, the base for FORTRAN is extremely strong.

In the realm of microcomputers, BASIC software currently predominates. The spectrum of applications includes both business data processing and some mathematical software. BASIC software, however, is likely to be more dependent on a particular system or implementation than the languages just mentioned, because of the slow pace of standardization.

Pascal and PL/I offer less support in the area of existing software, probably because the major application areas were pre-empted by the earlier languages. Some Pascal software is available, however, in the microcomputer arena.

2.1.10 Software Development Support -

A very important consideration in comparing languages is the availability of software tools and features which assist programmers in the construction and maintenance of programs. Although some of these features are embedded in the languages themselves, they are discussed here because they do not affect the logical behavior or the performance of operational programs; rather, their main purpose is to facilitate the human process of software development and maintenance. Software development support can also be implemented as part of a compiler or as an external package - see [Houg82], [Shah82], and [NBS83]. An external tool may apply to only one language, or it may be language-independent.

It is difficult to present a detailed comparison of the languages with respect to availability of software aids. That availability changes over time, and depends strongly on the vendor - neither factor a property of the language per se. The following generalizations give an overall perspective, but users are advised to make detailed inquiries when assessing support for a given language.

Ada, as a new language, is not yet strongly supported by software tools, but it is a central part of the Department of Defense plan for Ada to sponsor the development of a broad range

of such tools. It is a reasonable expectation that software support will become widespread as Ada is adopted and implemented.

Many microcomputer implementations of BASIC treat the language as an integral part of the system, and as such offer editing and debugging support (see below).

COBOL and FORTRAN, benefitting from their position as older standardized languages, have a wide variety of commercially available software support, although this is very often vendor-dependent. GSA/FSTC offers some vendor-independent tools to Federal agencies for COBOL.

Software development support for C, Pascal, and PL/I is less extensive than for the other languages. Particular operating systems may offer extra support, however. For instance, there are several C-oriented facilities in UNIX, and MULTICS provides tools for PL/I.

2.1.10.1 Source Code Manipulation And Checking -

This category covers all facilities which are directly involved in the development and modification of source code. Some languages have built-in features to incorporate pre-written sections of code. This is extremely useful for implementing such practices as standard data descriptions for files, etc. COBOL's COPY feature has long been used for this purpose. C's "#include" and PL/I's "%INCLUDE" provide similar capability. Ada's package facility (see 2.1.2.3, above), while more comprehensive than mere source inclusion, nonetheless can perform much the same function. C's "#define" statement and COBOL-8x's REPLACE statement cause systematic substitution of tokens within the source code. For instance, an identifier can be uniformly changed throughout the program.

External to the languages are such features as editors, prettyprinters, and library managers. The UNIX command "cb", for example, performs prettyprinting for C programs. Library managers are especially useful in large development projects involving many programs. They help keep track of such matters as which versions of code have been compiled, which modules depend on others, etc. It is anticipated that a sophisticated library management system will be available for Ada. This is reflected in the standard's requirements regarding compilation. Language-based editors are a relatively new and extremely promising development. Such editors incorporate many of the syntax rules of the language and thus are capable of automatically generating syntactically correct code in response to user commands. Typically, the code produced will also follow built-in conventions for indenting and grouping of entities, i.e. the usual prettyprinter functions.

Finally, tools are available which check source code for adherence to language standards and conventions. Some tools allow individual installations to establish and enforce their own conventions. Others, often built into compilers, monitor code for conformance to the ANSI standard. The FIPS for COBOL and FORTRAN require that vendors provide this facility with their language processors when selling to the Federal Government. The ANSI standards for Ada and Pascal also require implementations to be capable of syntax-checking. In the UNIX environment, the program "lint" performs a portability check on C programs. Finally, there is a utility [Hopk83] to do the same for Minimal BASIC (not the proposed full version).

2.1.10.2 Program Testing -

Software aids in the area of testing come from all three sources of software tools: language-embedded, compiler-embedded, and external. These aids are used to detect syntax and logic errors in programs, but not to generate correct code.

Only two languages, BASIC and COBOL, have debugging statements. The effect of these statements can be turned off and on, with a run-time switch in BASIC and a compile-time or run-time switch in COBOL. They allow the user to trace the effect of execution, and so help diagnose problems. C has a related feature, conditional compilation, which allows the user to tell the compiler whether to ignore or compile sections of code; clearly, this can be of use in debugging.

As mentioned earlier, software development features usually depend on the vendor, rather than the language. BASIC is unique, however, in that it is very often implemented with an interpreter. It is then quite common to find interactive debugging provided. Such features as being able to interrupt and resume execution, or to pause and display the current contents of variables are typical in an interpretive environment.

Compiler and run-time diagnostics can be of great value when testing code. [Shah82] covers many of the common features. Especially for those languages without exception-handling (section 2.1.2.1.7), it is important that the system provide useful information when encountering anomalous run-time conditions. Some vendors provide debugging facilities, often geared to interactive program development, which apply to all the languages implemented on their system. Of course, compile-time diagnostics should also be easy to understand and should be helpful in identifying problems. There are some rather sophisticated tools available for such functions as analysis of system dumps, static and dynamic control flow analysis, test coverage, etc. See [Houg82] for a full description. Most of these tools are oriented to FORTRAN or COBOL.

2.1.10.3 Information And Analysis -

Compilers and external utilities can also be useful in analyzing both the logical and performance characteristics of programs. Such facilities as cross-reference tables for identifiers, statistics on the uses of different types of statements, object code listings, and statistics on storage and time usage are all helpful in solving logic problems, and analyzing performance. Again, it is difficult to draw any general conclusions about the availability of such tools for the various languages, other than to point out the predominance of COBOL and FORTRAN as objects of such facilities.

2.2 Application Requirements (see Figures 15 And 16)

Now that we have examined what the languages under study have to offer, we shall look at the criteria for analyzing the application requirements. We shall proceed from those criteria most closely bound up with the logical definition of the application in question, to those determined by the environment in which it is to be implemented. The description of each criterion will refer back to those language features most relevant to that requirement. The overall relationship between application requirements and language factors is summarized in Figure 15.

2.2.1 Functional Operations -

The most basic requirement is the ability to perform the operations involved in the application. For instance, does the application use fixed-point or floating-point calculation? Is there a great deal of character string manipulation? Will the application need interactive graphics? These requirements are most commonly expressed by characterizing the application as "business-oriented" (implying fixed-point decimal arithmetic, character manipulation, and file-handling) or "scientific" (implying array manipulation and numerical calculation). While these descriptions have validity for some applications, they are often oversimplifications. Systems analysts and designers should carefully determine the full array of needed operations and functions, and not rely too heavily on the use of two or three simple categories.

The language features most relevant to these requirements are the data types and application facilities. When no one language appears adequate to the task, users should consider a multi-language approach - although there are associated costs, especially because of the lack of standardization of inter-language capabilities. Bearing in mind their limitations, we can make some broad generalizations about the suitability of the various languages for certain classes of application.

Figure 15 - Language Factors vs. Application Requirements
(see section 2.2)

This figure illustrates which language factors are most relevant in fulfilling the various application requirements. "XX" indicates that the factor is strongly supportive of the requirement; "X" indicates a less important factor in satisfying the requirement.

	LANGUAGE FACTORS								
APPLICATION REQUIREMENTS	Semantic Structure	Data Types and Manipulation	Application Facilities	Program Implementation Control	Simplicity	Standardization	Performance	Software Availability	Software Development Support
Functional Operations		XX	XX						
Size and Complexity	XX				X			X	XX
Number of Programmers	XX					XX			XX
Expertise	XX	XX	XX		XX	X			
End-user Interaction	X	XX					X		
Reliability	XX	X			XX				XX
Timeframe	XX				XX	XX	X	XX	XX
Portability						XX			
Execution Efficiency				X			XX		

Figure 16 - Languages vs. Application Requirements (see section 2.2)

APPLICATION REQUIREMENTS	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
Functional Areas: Business		X		XX			XX
Math, Science		X			XX		XX
Systems	XX		XX				X
Real-time	XX	XX					
Education		XX			X	XX	
Size and Complexity	Large	Small	Moderate	Large	Small	Moderate	Large
Number of Programmers	XX			XX			X
Expertise	Expert	Casual	Moderate	Expert	Casual	Moderate	Expert
End-user Interaction	X	XX	X		X	X	X
Reliability	XX	XX				X	X
Timeframe	Long	Short	Moderate	Long	Short	Long	Long
Portability	XX		X	XX	XX	XX	
Execution Efficiency	X		XX		XX	X	

Key: XX - strong support
X - moderate support

For information processing applications (business-type), COBOL is clearly the language of choice. BASIC and PL/I, as well as COBOL, support decimal arithmetic, character manipulation, and sophisticated file-handling, and are therefore also reasonable choices. BASIC's lack of a record construct (see 2.1.3.3.3) hampers its use, especially for more complex applications.

FORTRAN offers the strongest support for scientific, mathematical, and engineering applications, especially in the extensive software available for purchase. PL/I and BASIC should be considered, in that they have a large assortment of built-in functions, extended precision floating-point numbers, and array manipulation. PL/I supports complex numbers as well.

Ada, C, and PL/I are suitable for systems programming, in that they are typically implemented so as to allow fairly direct access to the underlying machine, although only Ada directly supports parallel processes (tasking).

Real-time applications (other than systems programming) may be handled by Ada or BASIC. If the proposed standard for Industrial Real-Time FORTRAN (see section 2.1.4.3) becomes widely accepted, FORTRAN would also be a sensible option.

For educational applications, it is important that a language allow clear and simple expression of the concepts being taught. BASIC, for elementary concepts of computers and programming, and Pascal, for more theoretical topics, are the most suitable choices. Of course, education was the original design goal of both languages.

2.2.2 Size And Complexity -

This criterion is concerned with the size and complexity of the application (and presumably, therefore, of the programs), to be implemented. While size and complexity are not necessarily linked, they are discussed together because the same language features which help to manage one also help to manage the other. Size is self-explanatory - number of lines of code may be taken as an adequate metric. An application may be complex as a result of especially sophisticated algorithms or data structures, or because modules within the application interact in a large number of ways. Real-time applications tend to be more complex than those with sequential control. Similarly, the updating of master files and databases is more complex than read-only operations.

For large, complex applications, language features which support the creation and maintenance of code (software availability and development support) or help the programmer to express complex relationships (program structure) are especially relevant. Ada is noteworthy as a language which had the development of large, complex systems as an explicit design goal, and many features of Ada reflect that goal. PL/I also provides

useful features for attacking large applications. COBOL is strong in the variety of software tools commercially available for managing development and maintenance.

Conversely, for small (< 250 lines), simple applications, it is desirable to use a language which does not incur an unnecessarily large syntactic and conceptual overhead. Simplicity therefore becomes a crucial property. BASIC and FORTRAN are good choices for such applications. Pascal and C are well-suited for intermediate sized programs.

2.2.3 Number Of Programmers -

To the extent that a program or system is to be developed or maintained by several individuals, it becomes important that the code be readily understandable. Language features in the categories of program structure, standardization, and software development support are clearly relevant. The strong languages in this area are Ada and COBOL. PL/I also has helpful features, but suffers from weak standardization.

2.2.4 Expertise -

Unless an agency has the option of hiring new programmers, it must reckon with the skills of its current staff. Using an unfamiliar language will incur costs, either for training or for having the work performed on contract. It is important to balance short-term costs against potential long-term gains. If there is likely to be an ongoing need for programmers skilled in a given language, training can be a sound investment. And, of course, some languages are much easier to learn than others.

Managers should also take into account whether the application is to be implemented by professional programmers, i.e., those whose primary job is programming, or by casual programmers - those for whom programming is a secondary skill. With the spread of microcomputers, casual programming is becoming more common. Such programming is best done in a relatively simple language, but also in a language which provides application facilities directly to the user, since casual users should not be expected to construct their own from more primitive features. BASIC and FORTRAN are usually the best choices.

Professional programmers typically can make good use of more sophisticated features. Ada, COBOL, and PL/I are more oriented to this type of practice. Pascal and C are intermediate cases; C, although relatively simple, is appropriate for systems programmers. Typically, programming in C involves the construction and use of sophisticated libraries of system functions from C's simple repertoire of features.

If new programmers are to be hired, then the available pool of programmers skilled in a language becomes an important consideration. One key advantage to strong standardization is the creation of such a reservoir of expertise. Currently, COBOL and FORTRAN are most widely known among professional programmers. Pascal and BASIC are being used in educational environments, and hence are also well-known.

2.2.5 End-user Interaction -

Batch processing can be equally well handled by all the languages. Interactive processing, however, requires language features to support that type of I/O operation. Also, it is important that the run-time performance be adequate for quick response to user actions.

BASIC strongly supports interactive applications, both in its I/O and in having graphics capabilities. Also, the proposed standard defines helpful run-time recovery in case of invalid input. COBOL does not provide good I/O support for free format data. Pascal also is weak in this regard. All the other languages can perform reasonably well.

2.2.6 Reliability -

Certain applications perform critical functions which require a great deal of reliability. Many real-time applications fall into this class. An application involving the transfer of large sums of money is another example. Program structure features, especially exception-handling, help meet this requirement. Type checking is also thought to be useful in early detection of certain kinds of error. Software development support and language simplicity can contribute to the construction of correct programs.

As with size and complexity, reliability was one of the major design goals for Ada, and consequently Ada offers the strongest support for this goal, although its complexity may contribute to errors among less experienced programmers. BASIC has only a few simple data types and exception handling, so it too is a good choice. Pascal offers strong type checking, but no exception handling; PL/I offers exception handling but very little type checking.

2.2.7 Timeframe -

Some programs are written, executed a few times, and then discarded. Others become part of a system which may be operational for decades. Clearly, ease of writing is a dominant concern in the first case, and ease of reading in the second. Even for long-lived systems, however, development requirements may differ. Sometimes there is a need for prototyping to get some operational capability quickly; or, schedules may allow more time for pre-implementation design. Program structure, simplicity, software availability, and software development all pertain to these goals. Also, the trade-off between compilation and run-time performance will affect the two cases differently. Finally, strong standardization is much more important to a system with a long lifetime than to a short-term effort, since it is very possible that the system will outlive the hardware on which it is originally implemented.

For rapid development, BASIC, FORTRAN, and to a lesser extent, C, are appropriate, with their emphasis on simplicity and low syntactic overhead. Also, users can take advantage of the considerable body of existing software available in these languages. For long-lived systems, Ada, COBOL, Pascal, and PL/I are usually better choices, with their emphasis on more detailed declarations and elaborate compilation.

2.2.8 Portability -

Portability is used here in the narrow sense of being able to re-use source code on different systems. A typical case in which there is a strong requirment for portability would be a central design agency writing code to be run at various field installations. Of course, the achievement of portability always depends strongly on proper management and coding practices on the part of the user; the language alone cannot guarantee such a result.

Clearly, standardization is the key criterion for this requirement. At present, the strongest language standards are for Ada, COBOL, FORTRAN, and Pascal; in the case of C, there is relatively strong <u>de facto</u> standardization.

2.2.9 Execution Efficiency -

Some applications place heavy demands on hardware resources once they are in operation. Examples are systems with high transaction rates, very large files or databases, and lengthy numerical calculations. Meeting such requirements is largely a matter of adequate hardware, but language performance and the ability to guide compilers to efficient implementation are also helpful. Ada, C, FORTRAN, and Pascal are likely to yield more

efficient execution for internal (non-I/O) operations, although this varies from one implementation to another. For I/O efficiency, there is little indication of major differences among the languages.

2.3 Installation Requirements

The last set of requirements are those imposed, not by the application to be implemented, but by the existing and potential resources of the installation doing the implementation. Very few installations are able to consider each project de novo; more often, new systems must evolve smoothly from existing practice. Nonetheless, DP managers should not neglect the possibility of reaping long-term ongoing benefits simply to avoid some short-term costs.

2.3.1 Language Availability -

The most basic issue is whether the installation has access to a processor for a given language. Let us first consider the traditional environment with one central large- or medium-size machine. Given a project of any size, the cost of an additional compiler can usually be justified if the language offers substantial benefits over those already available. Language processors may be obtained either from the hardware vendor or from independent software firms, or accessed through timesharing. Sometimes implementation includes the procurement of hardware, and in these cases, language availability should be an important factor in that decision. Very often, however, language selection will be limited to those already available on an existing system.

The advent of microcomputers poses some new issues in the choice of languages. It is much more likely that implementation will include hardware acquisition, as compared to the use of larger machines. The cost (per system, at least) of hardware and software is likely to be relatively low - hence the choice among languages can be a freer one. As with the procurement of large systems, language availability for a proposed microcomputer should be carefully considered before purchase. The microcomputer environment has been dominated by BASIC, C, Pascal, and to some extent FORTRAN. It is no longer unusual, however, to see implementations of COBOL or PL/I subsets, and at least one complete implementation of COBOL exists. Ada implementations, of course, are not as yet widespread in any class of machine, but some are being developed for microcomputers as well as for large machines.

2.3.2 Compatibility With Existing Software -

Very often, new systems must interact with existing software. Clearly, this interaction is simplified when the same language is used. Problems may arise when two parts of a system are written in different languages. The two most common ways programs of different languages interact are 1) for one program to call another as a subroutine, and 2) for one program to read files written by another. Unfortunately, there are no inter-language standards addressing these two areas, and users must ensure that the actual implementations of the languages they will employ are compatible for the intended purpose. Note that Ada has an INTERFACE pragma, and COBOL an ENTER statement, whose purpose is to allow communication with "foreign" code.

3.0 LANGUAGE SUMMARY (SEE FIGURE 17)

In this section, we shall recapitulate briefly the topics covered above for the various languages and point out their special advantages and disadvantages. Also, the committee(s) responsible for standardization are described. Volume 2 contains program examples to help illustrate the style and approach each language encourages.

3.1 Ada

The major design goal of Ada is the ability to handle large, real-time applications, with a premium on reliability and portability. The whole structure of the language is driven by these requirements. Ada's tasking facilities and representation clauses support efficient real-time operations. The extensive data-type checking and exception-handling support reliability. Ada's modularity features, such as packages and generics, support the construction of large systems. And finally, the strong commitment to standardization promotes portability. The result is a language which is very well suited for that class of applications.

Ada is clearly a "professional's" language. There is significant overhead when learning the language or writing a program. Using Ada, the programmer has the power to construct complex data structures and algorithms. Ada is a big language for big jobs. Rather than build many facilities into the language, the choice was made to rely on Ada's extensibility. Thus, the construction of application-specific packages is a prerequisite to making good use of Ada in areas beyond those originally envisioned.

The Ada Joint Program Office (AJPO) within the Department of Defense is the sponsor for Ada and is primarily responsible for development and maintenance of the language. Ada was adopted under the canvass method of the ANSI procedures. AJPO has set up the Ada Validation Organization (AVO), which performs validation testing on candidate Ada implementations.

Ada Joint Project Office
3D139 (400AN)
The Pentagon
Washington DC 20301

3.2 BASIC

BASIC was originally designed as a vehicle for teaching students elementary concepts of computers and programming. The goal was that meaningful results could be obtained with very little detailed knowledge of the language or machine.

Figure 17 - Languages and Standards Bodies (see section 3.0)

PARENT ORGANIZATION	Ada	Proposed BASIC	C	COBOL	FORTRAN	Pascal	PL/I
ANSI/X3		X3J2	X3J11	X3J4	X3J3	X3J9	X3J1
ISO/TC97/SC5*	WG14	WG13		WG8	WG9	WG4	WG11
ECMA		TC21		TC6	TC8		
EWICS		TC2			TC1		
other	Ada Joint Program Office			CODASYL COBOL Committee		IEEE/CS Pascal Standards Committee	

* see Appendix B.5 concerning re-organization

Furthermore, the language should provide meaningful diagnostic messages in response to mistakes. These original goals have been enlarged to include the incorporation of facilities in the language to support the most common applications: data processing and numeric calculation. Nonetheless, the language has remained simple in concept; for instance, by default there are only two data-types, variable-length character strings and decimal floating-point numbers.

The result is a language for end-users: those for whom programming is a secondary skill. BASIC's emphasis on good diagnostics, run-time checking, and meaningful defaults all support such use. The language is therefore an appropriate tool for relatively simple interactive applications. BASIC remains a good teaching tool at the introductory level; clearly it is not meant to embody more advanced concepts of computer science, but rather to provide a simple accessible way for beginners to learn about programming.

The draft standard for BASIC has been developed jointly by three committees, ANSI/X3J2, ECMA/TC21, and EWICS/TC2. Responsibility for BASIC within ISO is held by TC97/SC5/WG13.

3.3 C

The main design goal of C is to enable relatively portable systems programming. Consequently, C is a lower-level language than the others considered here. The entities of the language correspond closely to typical existing hardware, rather than to more logical, problem-oriented concepts. Thus, the absence of record I/O and the inclusion of bit operations on machine words (presumably equivalent to the integer type), register variables, and incrementing and decrementing operations.

C would normally not be the best choice for applications programming. Conversely, no other language is likely to be both as portable and efficient for the development of systems software, such as I/O drivers, compilers, utilities, etc. In particular, C should be used wherever possible if the only alternative is assembler language.

Formal standardization work on C started only recently. ANSI committee X3J11 is currently developing a draft specification.

3.4 COBOL

COBOL is, of course, the pre-eminent language for data processing (business-type) applications, as it was designed to be. Its support for file processing and editing is very strong. As a secondary benefit, its very dominance in the DP arena

assures that a great deal of COBOL-oriented commercial software will be available. There are weaknesses in the language; in particular it provides little support for interactive I/O and its substring manipulation features are less extensive than might be expected. Nonetheless, it is likely to remain the dominant business language for the foreseeable future. While COBOL is often thought of as a "main-frame" language, implementations are becoming increasingly available on microcomputers. Many of these compilers conform to one of the lower levels of the standard [NBS75], and users should be aware of which features are included and excluded by a subset under consideration for purchase.

The development of COBOL is under the purview of the CODASYL COBOL Committee, which publishes the proposed language specification in the CODASYL COBOL Journal of Development. ANSI committee X3J4 has primary responsibility for standardization of the results of this development activity. Internationally, TC97/SC5/WG8 and ECMA/TC6 contribute to the development and maintenance of COBOL. CODASYL's address is:

CODASYL: CBEMA
311 First Street NW, Suite 500
Washington, DC 20001

3.5 FORTRAN

As COBOL dominates business applications, so FORTRAN has come to dominate scientific, engineering, and mathematical applications. FORTRAN offers a broad array of functions, complex numbers, and perhaps most important, a very substantial base of software. FORTRAN is the oldest of the languages under study, and its age is reflected in several shortcomings, notably its weak support for software maintenance. Also, there are no language features to facilitate processing of arrays and no recursive procedures, surprising omissions for a computational language. While FORTRAN is quite suitable for applications of moderate size, users should consider other languages when embarking on a major development effort. FORTRAN is available on a wide variety of machines. Although most implementations follow the standard, some do not. Since the standard ensures support for character string manipulation and some structured programming, users are well-advised to stay with standard-conforming processors.

ANSI committee X3J3 has primary responsibility for language development and maintenance. Internationally, TC97/SC5/WG9, ECMA/TC8, and EWICS/TC1 all contribute to the FORTRAN effort.

3.6 Pascal

Pascal was designed as a vehicle for teaching programming and computer science. Consequently, the language structure reflects a concern for conceptual consistency and clarity. Pascal has become quite successful in fulfilling this design goal; it is probably the most common language used in college level computer science courses. Certain features used in common applications, but which are less important pedagogically, were omitted, e.g., string-handling, enhanced numeric data-types and operations, and external procedures. Thus, Pascal is not especially well suited to DP or scientific applications. Moreover, the language is complex enough that it is not appropriate for casual use. Pascal has been used successfully for certain classes of systems programming, such as parsers, which are not strongly dependent on the underlying hardware.

There is active effort on Pascal standardization in both the national and international arena. Nationally, ANSI/X3J9 and the IEEE Pascal Standards Committee have combined to form the Joint Pascal Committee. Within ISO, TC97/SC5/WG4 has responsibility for Pascal.

Within North America, the authorized distribution agent for the Pascal Validation Suite developed by Professor Arthur Sale of the University of Tasmania, and others, is:

Software Consulting Services
Ben Franklin Technology Center 125
Murray H. Goodman Campus
Lehigh University
Bethlehem PA 18015
(215) 861-7920

The British Standards Institution offers a Pascal Validation Service, which is based on the ISO standard.

3.7 PL/I

PL/I was designed to be a truly general-purpose programming language: one which would support the great majority of end-user and systems applications. Thus, the language contains a large number of features: a wide assortment of data-types, strong file support, and detailed control of storage. Moreover, all the structuring features of modern languages are implemented, e.g., blocks and recursion. PL/I is certainly capable of handling the applications it was designed for. Because its features are built-in and standardized, the language can be used for writing portable programs, without recourse to libraries of user-defined routines. Especially in the case of applications which overlap several of the traditional categories (business, scientific, systems), the breadth of PL/I provides a unique advantage. Because the language is so powerful, it is also large and

complex. The rules for defaults and data conversion are difficult to master. Somewhat like Ada, PL/I is a powerful language for use by professional programmers in large applications. For simpler applications, simpler tools exist.

Unfortunately, the full PL/I standard has not been widely implemented, so added to the complexity of the language itself is the variation among compilers. As mentioned earlier, the PL/I subset standard [PL/I81] has achieved broader acceptance. ANSI X3J1 and SC5/WG11 are the committees primarily responsible for PL/I standards.

4.0 CONCLUSION

When an application is to be implemented with conventional programming, the choice of language can have a major effect on the success of the project. Moreover, the user must carefully consider the costs and benefits not only during development, but also throughout the life of the application. In many cases, maintenance costs exceed development costs. While it is not possible to formulate a precise method for choosing the best language, a review of the criteria presented in this report will help at least to avoid the worst choices.

It can hardly be emphasized too strongly that users should not ignore long-term costs and benefits. For small short-term projects, the total risk is low in any case. But for larger projects, many indirect criteria may become crucial. In particular, it can be a decisive advantage when a language is supported by strong standardization.

REFERENCES

[Ada83] American National Standard Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, American National Standards Institute, New York NY, 1983.

[Baas78] Baase S., Computer Algorithms: Introduction to Design and Analysis, Addison-Wesley, Reading MA, 1978.

[BASI84] Draft Proposed American National Standard for BASIC, X3J2/84-26, X3 Secretariat: Computer and Business Equipment Manufacturers Association, Washington DC, June 1984

[Byte83] Byte, Vol. 8 No. 8, August 1983. This issue has several useful articles on C and its implementations.

[Byte84] Byte, Vol. 9 No. 8, August 1984. This issue has several useful articles on Modula-2.

[COBO74] American National Standard Programming Language COBOL, ANSI X3.23-1974, American National Standards Institute, New York NY, 1974.

[COBO83] Draft Proposed American National Standard Programming Language COBOL, BSR X3.23-198X, X3 Secretariat: Computer and Business Equipment Manufacturers Association, Washington DC, 1983.

[Comp83] Computers & Standards, Vol. 2 No. 2-3, 1983. This issue is devoted entirely to the current state of programming language standardization.

[Feue82] Feuer A. R. and Gehani N. H., "A Comparison of the Programming Languages C and Pascal", ACM Computing Surveys, Vol. 14 No. 1, March 1982.

[Feue84] Feuer A. R. and Gehani N. H., Comparing and Assessing Programming Languages, Prentice-Hall, Englewood Cliffs NJ, 1984. Collection of articles with detailed comparison and evaluation of Ada, C, and Pascal. Also contains articles on methodology of language comparison.

[FORT78] American National Standard Programming Language FORTRAN, ANSI X3.9-1978, American National Standards Institute, New York NY, 1978.

[Fran84] Frankel S., Introduction to Software Packages, NBS Special Publication 500-114, National Bureau of Standards, Gaithersburg MD, April 1984.

[FSTC84] Certified Compiler List, Report OIT/FSTC-84/004, Federal Software Testing Center, Falls Church VA, July 1984.

[Gall84] Gallagher L. J. and Draper J. M., Guide on Data Models in the Selection and Use of Database Management Systems, NBS Special Publication 500-108, National Bureau of Standards, Gaithersburg MD, January 1984.

[Ghez82] Ghezzi C. and Jazayeri M., Programming Language Concepts, John Wiley & Sons, New York NY, 1982. Very up-to-date, emphasizes software engineering and implementation issues, as well as language design. Glossary gives overview of 20 languages. Detailed programming examples in Ada, ALGOL 68, APL, LISP, Pascal, and SIMULA 67.

[Hals77] Halstead M. H., Elements of Software Science, Elsevier - North Holland, New York NY, 1977

[Hech84] Hecht M., Hecht H., and Press L., Microcomputers: Introduction to Features and Uses, NBS Special Publication 500-110, National Bureau of Standards, Gaithersburg MD, March 1984.

[Hech8x] Hecht M., Hecht H., and Press L., Microcomputer Applications Programs for Software Development, National Bureau of Standards, Gaithersburg MD, to be published.

[Hill80] Hill I. D. and Meek B. L., Programming Language Standardisation, Ellis Horwood Limited, Chichester UK, 1980. Good overview of the history of and procedures for international promulgation of language standards. Chapters on ALGOL 60, BASIC, COBOL, FORTRAN, Pascal, and PL/I.

[Hopk83] Hopkins T. R., "Algorithm 605 PBASIC: A Verifier Program for American National Standard Minimal BASIC", ACM Transactions on Mathematical Software, Vol. 9 No. 4, December 1983.

[Horo84] Horowitz E., Fundamentals of Programming Languages, 2nd edition, Computer Science Press, Rockville MD, 1984. Good overview of current language concepts and issues, including less conventional topics such as exception-handling, concurrency, functional programming, data-flow programming, and object-oriented programming. Discusses Ada, ALGOL, APL, CLU, Euclid, FORTRAN, LISP, MESA, MODULA, Pascal, PL/I, SIMULA, Smalltalk, SNOBOL, and VAL.

[Houg82] Houghton R. C., Software Development Tools, NBS Special Publication 500-88, National Bureau of Standards, Gaithersburg MD, March 1982.

[IRTF84] Industrial Real-Time FORTRAN - Application for the control of industrial processes, Draft International Standard, ISO/DIS 7846, ISO/TC97/SC5 Secretariat: American National Standards Institute, New York NY, 1984.

[ISO83] Programming languages - Pascal, ISO 7185-1983, British Standards Institution, London UK, 1983.

[ISPS81] Information Systems Planning Service, Impact of the Newer Programming Languages, ISPS-M81-03, International Data Corporation, Framingham MA, March 1981. Short survey paper covering Ada, ALGOL, APL, C, COBOL, Forth, FORTRAN, Pascal, and PL/I.

[ISPS84] Information Systems Planning Service, New Programming Languages, IDC #2483, International Data Corporation, Framingham MA, May 1984. Short survey paper covering Bliss, CLU, Concurrent Pascal, Dibol, Euclid, Jovial, Lisp, Logo, Mesa, Modula-2, MUMPS, PROLOG, Simula 67, Smalltalk, and SNOBOL.

[Jali84] Jalics P. J., "COBOL vs. PL/I: Some Performance Comparisons", Communications of the ACM, Vol. 27 No. 3, March 1984.

[Jens74] Jensen K. and Wirth N., Pascal User Manual and Report, Springer-Verlag, New York NY, 1974.

[Kern78] Kernighan B. W. and Ritchie D. M., The C Programming Language, Prentice-Hall, Englewood Cliffs NJ, 1978.

[Mart82] Martin J., Application Development without Programmers, Prentice-Hall, Englewood Cliffs NJ, 1982.

[MacL83] MacLennan B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart and Winston, New York NY, 1983. Discusses design and implementation issues in a thorough, practical, and clear manner with many useful examples. Excellent explanations of newer languages, such as FFP, Smalltalk, and PROLOG. Also covers Ada, ALGOL-60, FORTRAN, LISP, and Pascal.

[McGe80] McGettick A., The Definition of Programming Languages, Cambridge Univerisity Press, Cambridge UK, 1980. Complete discussion of the technical problems of defining (and hence standardizing) syntax and semantics of programming languages. Covers ALGOL 60, ALGOL 68, ALGOL W, BASIC, COBOL, FORTRAN, LISP, Pascal, and PL/I.

[NBS75] COBOL, FIPS PUB 21-1, National Bureau of Standards, Gaithersburg MD, December 1975.

[NBS80] Minimal BASIC, FIPS PUB 68, National Bureau of Standards, Gaithersburg MD, September 1980.

[NBS80a] FORTRAN, FIPS PUB 69, National Bureau of Standards, Gaithersburg MD, September 1980.

[NBS81] Interpretation Procedures for Federal Information Processing Standard Programming Languages, FIPS PUB 29-1, National Bureau of Standards, Gaithersburg MD, December 1981.

[NBS83] Guideline: A Framework for the Evaluation and Comparison of Software Development Tools, FIPS PUB 99, National Bureau of Standards, Gaithersburg MD, March 1983.

[Pasc83] American National Standard Pascal Computer Programming Language, ANSI/IEEE770X3.97-1983, Institute of Electrical and Electronics Engineers, New York NY, 1983.

[PL/I76] American National Standard Programming Language PL/I, ANSI X3.53-1976, American National Standards Institute, New York NY, 1976.

[PL/I81] American National Standard Programming Language PL/I General-Purpose Subset, ANSI X3.74-1981, American National Standards Institute, New York NY, 1981.

[Prat84] Pratt T. W., Programming Languages: Design and Implementation, 2nd edition, Prentice-Hall, Englewood Cliffs NJ, 1984. Very thorough, well-organized, up-to-date text. Discusses both design and implementation issues in detail. Covers Ada, APL, COBOL, FORTRAN, LISP, Pascal, PL/I, and SNOBOL4.

[Samm69] Sammett J. E., Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs NJ, 1969. Classic work, although now somewhat out-of-date. Practical orientation, thorough discussion of selection criteria. Covers ALGOL60, COBOL, FORTRAN, JOVIAL, LISP 1.5, PL/I, SNOBOL, and many others.

[Samm81] Sammett J. E., "An Overview of High-Level Languages", Advances in Computers, vol. 20, ed. Yovits M. C., Academic Press, New York NY, 1981.

[Shah82] Shahdad B. M. and Libster E., Compiler Features: A Survey, NBS-GCR-82-418, National Bureau of Standards, Gaithersburg MD, 1982.

[Tenn81] Tennent R., Principles of Programming Languages, Prentice-Hall, Englewood Cliffs NJ, 1981. Formal, mathematical approach, emphasizing Pascal, with some discussion of ALGOL68, APL, LISP, SIMULA, and SNOBOL.

[Tuck77] Tucker A. B., Programming Languages, McGraw-Hill, New York NY, 1977. Practical orientation, with detailed examples, selection criteria, and some performance evaluation. Covers ALGOL60, COBOL, FORTRAN, PL/I, RPG, and SNOBOL.

[Wass82] Wasserman A. I., "The Future of Programming", Communications of the ACM, Vol. 25 No. 3, March 1982.

[Vale74] Valentine S. H., "Comparative Notes on ALGOL 68 and PL/I", The Computer Journal, Vol. 17 No. 4, November 1974.

[Zveg83] Zvegintzov N., "Nanotrends", Datamation, Vol. 29 No. 8, August 1983.

ACKNOWLEDGMENTS

The following individual language experts graciously consented to review this publication for technical accuracy and general soundness of concept: John Caron, John Goodenough, John Klensin, John A. N. Lee, Brian Meek, Donald Nelson, Jean Sammet, and Donald Warren. Nils Brubaker, Al Deese, Brian Schaar, and Henry Tom rendered timely and valuable assistance in testing the program examples. All their efforts contributed strongly to this report. The responsibility for whatever errors remain rests of course with the author.

APPENDIX A

ABBREVIATIONS

ACM	Association for Computing Machinery, scientific and technical association with broad interest in computers, academic orientation.
AJPO	Ada Joint Project Office, agency within Department of Defense with primary responsibility for Ada standards, sponsor of Ada as ANSI standard.
ANSI	American National Standards Institute, organization fostering voluntary national standards.
AVO	Ada Validation Organization, set up by AJPO to administer validation of Ada implementations.
BSI	British Standards Institution, organization fostering voluntary national standards in the United Kingdom, offers Pascal Compiler Validation Service.
CBEMA	Computer and Business Equipment Manufacturers Association, American trade association, provides secretariat for X3.
CODASYL	Conference on Data System Languages, committee responsible for the development (but not standardization) of COBOL specifications.
ECMA	European Computer Manufacturers Association, European trade association, participates actively in ISO/TC97 standardization activities.
EWICS	European Workshop on Industrial Computer Systems, organization concerned with language control of real-time systems.
FIPS	Federal Information Processing Standard, authorized by the Department of Commerce to manage information processing activities within the Federal Government, developed and issued by ICST/NBS.

FSTC	Federal Software Testing Center, within GSA, administers validation tests for FIPS languages.
GSA	General Services Administration, responsible for property management within the Federal Government, parent body of FSTC.
ICST	Institute for Computer Sciences and Technology, administers FIPS program for the Federal Government, assists Federal agencies, performs research in computers and networks.
IEEE/CS	Institute of Electrical and Electronics Engineers Computer Society, professional association with broad interest in computers.
ISO	International Organization for Standardization, organization fostering voluntary international standards.
NBS	National Bureau of Standards, agency with primary responsibility for measurement methods, standards, and data for physical and engineering sciences within the Federal Government, parent body of ICST.
NTIS	National Technical Information Service, sells technical products developed for and within the Federal Government.
SC5	Subcommittee 5 - Programming Languages, of ISO/TC97, has primary responsibility for language standards within ISO. Individual languages handled by working groups (WG's) within SC5 (see Appendix B.5 concerning re-organization).
TC97	Technical Committee 97 - Information Processing Systems, has primary responsibility for computer standards within ISO, parent body of SC5 (see Appendix B.5 concerning re-organization).
X3	X3 - Information Processing Systems, the American National Standards Committee for computer standards operating under the procedures of ANSI. Individual languages handled by "J" technical subcommittees, e.g., X3J1 for PL/I.

APPENDIX B

SOURCES OF INFORMATION

In addition to the organizations mentioned in section 3 which work with individual languages, there are several bodies with a general interest in programming languages and computer standards. Below is a brief description of the areas of concern and mailing address and telephone number for the most prominent of these bodies.

B.1 INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY

The Institute for Computer Sciences and Technology (ICST) within the National Bureau of Standards (NBS) has responsibility for Federal Information Processing Standards, providing technical assistance to Federal agencies, and conducting research in computer and network technology. ICST participates actively in voluntary industry standards development activities, including programming languages. Within ICST, the Center for Programming Science and Technology is responsible for programming language standards.

Institute for Computer Science and Technology
Center for Programming Science and Technology
Data Management and Programming Languages Division
Building 225, Room A-255
National Bureau of Standards
Gaithersburg, MD 20899
(301) 921-2431

B.2 FEDERAL SOFTWARE TESTING CENTER

The Federal Software Testing Center (FSTC) within the General Services Administration (GSA) participates in the administration of the GSA procurement regulations for the Federal Government. In particular, FSTC maintains and applies validation systems for the various languages approved for Federal use (currently COBOL, FORTRAN, and Minimal BASIC). Also, FSTC maintains a register [FSTC84] of implementations which have

undergone this validation process and are thus eligible for Federal procurement. This register includes Ada compilers.

Federal Software Testing Center
Two Skyline Place, Suite 1100
5203 Leesburg Pike
Falls Church, VA 22041
(703) 756-6153

B.3 NATIONAL TECHNICAL INFORMATION SERVICE

The National Technical Information Service (NTIS) within the Department of Commerce serves as a clearinghouse for technical publications developed for and within the Federal Government. ICST publications and software products are normally available for purchase through NTIS.

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
(703) 487-4600

B.4 X3 - INFORMATION AND PROCESSING SYSTEMS

X3 is an American National Standards Committee operating under the procedures of the American National Standards Institute (ANSI). There are technical subcommittees within X3 (see Figure 15) for all the languages in this report except Ada. An ANSI standard is voluntary. Participation by all those concerned with standards (producers, consumers, and others) is encouraged. The X3 secretariat is held by the Computer and Business Equipment Manufacturers Association (CBEMA).

X3 Secretariat: CBEMA
311 First Street NW, Suite 500
Washington, DC 20001
(202) 737-8888

B.5 SC5 - PROGRAMMING LANGUAGES

SC5 is a subcommittee of the International Standards Organization's (ISO) TC97 - Information Processing Systems. It is the body which co-ordinates international development and approval of language standards. The secretariat for ISO/TC97/SC5 is currently held by ANSI. At the time of this writing, TC97 is being re-organized, and the language standardization work is being assigned to two new subcommittees, SC21 - Information Retrieval, Transfer, and Management for OSI and SC22 - Application Systems Environments and Programming Languages.

SC5 Secretariat: ANSI
1430 Broadway
New York, NY 10018
(212) 354-3347

B.6 IEEE COMPUTER SOCIETY

While traditionally more concerned with hardware issues, the IEEE Computer Society (IEEE/CS) has recently taken a more active role in programming languages. It is co-operating with X3 (see above) on the standardization efforts for Pascal. Also, IEEE is proposing standards for floating-point arithmetic which will have important language implications.

IEEE Computer Society
1109 Spring Street, Suite 300
Silver Spring, MD 20910
(301) 589-8142

B.7 SPECIAL INTEREST GROUP ON PROGRAMMING LANGUAGES

The Special Interest Group on Programming Languages (SIGPLAN) of the Association for Computing Machinery (ACM) is a scientific and technical association devoted to the exploration of various language issues. Their informal publication, "SIGPLAN Notices", covers topics of current interest. ACM also publishes a formal quarterly journal, "Transactions on Programming Languages and Systems".

ACM Headquarters
11 West 42nd Street
New York, NY 10036
(212) 869-7440

APPENDIX C

ALTERNATIVES TO CONVENTIONAL PROGRAMMING

Historically, high-level programming languages made possible a great improvement in programmer productivity because they allowed the user to express algorithms and data structures in a comparatively problem-oriented way, as opposed to the hardware orientation of machine and assembler languages [Samm69]. There was, of course, a price to be paid: some loss in run-time efficiency and, perhaps more important, the inability to access directly all the hardware capabilities of a system. For instance, the FORTRAN programmer, as such, cannot do fixed-point decimal arithmetic, even if the hardware supports it. Nonetheless, no one would seriously argue today that assembler language should be used routinely for most applications; the gain in programmer productivity far outweighs the costs just mentioned.

Many experts believe that the practice of application development and maintenance is now on the verge of a transition comparable to that between assembler and high-level languages [Mart82], [Wass82]. Traditionally, the task of the programmer/analyst has been to devise algorithms that solve the problem presented by the end-user. The programmer figures out how to solve the problem and then expresses that solution in a programming language. Techniques are becoming available (see below) which will allow the programmer, or, better still, the end-user, to state more or less formally what results are desired. The system is then capable of producing these results automatically. At no time does a human need to generate an explicit algorithm. The new approach is usually described as "functional" as distinguished from "procedural" programming.

The analogies with the previous transition to high-level languages are apparent. Earlier, DP practitioners were freed from thinking about irrelevant hardware details; now they may be freed from thinking about irrelevant algorithmic details. For instance, the end-user is interested in updating the personnel file. The fine points of the associated merge algorithm are only incidental to solving the problem. And, as before, there is a price. It may be expected that there will be some degradation of run-time efficiency. Again, there will be a loss of "fine control" - users will depend on the stereotyped solutions built

into the new tools. Finally, just as high-level languages have not abolished assembler programming, but merely restricted it to those relatively few cases where machine efficiency and fine control are critical, it is safe to predict that there will be a continuing need for conventional programming for the foreseeable future, both for some new development, and certainly for maintenance of existing systems.

There is one aspect of this transition from conventional programming to functional techniques which is not analogous to the earlier transition from assembler language: almost all the popular languages are now standardized to some degree, whereas functional techniques are not. Thus a system written in (standard) COBOL does not depend in a crucial way on the support of any particular vendor, or on a given machine architecture. The same cannot be said (yet) of many of the functional approaches mentioned below.

When should one use conventional programming languages and when functional techniques? This issue is a matter of some debate; see [Mart82] for an articulate statement of the "pro-functional" position and [Zveg83] for a spirited rejoinder. A short answer is that the smaller, simpler, and more typical the application, the more susceptible it will be to the new functional techniques. Also, functional techniques tend to be associated with applications which are short-term or single-user or both, because of their relative advantages with respect to development time and required skills, and disadvantages with respect to machine efficiency and standardization. Because of the relatively short start-up time, the functional approach may also prove valuable when a system is to be designed with the aid of prototypes. Typically, the overhead of conventional programming is too great to allow the development of software only for the purpose of system design. On the other hand, insofar as fine control, standardization, and run-time efficiency are required, conventional programming is more likely to be the better approach. For example, a large, long-lived, logically complex system with high transaction rates would very likely not be amenable to functional implementation techniques.

There is as yet no precise way to evaluate the complex trade-offs involved in deciding between conventional programming and functional techniques. ICST plans to keep these issues under study and will issue more detailed guidance as experience with the new techniques accumulates. Those interested in the functional techniques should find the following publications helpful: [Hech8x], [Hech84].

The following sub-sections describe, very briefly, some of the alternatives to conventional high-level programming as a means of developing and maintaining applications. These descriptions should not be taken as a detailed guideline on usage and selection, but merely as an indication of the major approaches which are currently available.

C.1 DATABASE MANAGEMENT SYSTEMS

While it does not always constitute a complete application development method, the use of a database management system (DBMS) is often the basis for other techniques [Gall84]. For instance, an automatic report generator may presuppose the existence of a database from which the report is to be constructed. Of course, DBMS facilities may also be accessed from conventional programming languages. In either case, the establishment of an integrated database for a functional area will very often enable the use of more powerful software development technologies.

C.2 QUERY AND REPORT FACILITIES

Query and report facilities are perhaps the best established of the functional techniques. RPG and COBOL's Report Writer have been available for many years and provide good examples of the functional approach. The user specifies the information to be displayed and some indication of the desired format. The system generates the required report. At no time, for instance, does the user explicitly formulate the logic necessary to handle control breaks or page headings. Many database management systems have an associated query language that allows the user to display information from the database. Query and reporting are highly susceptible to a functional approach because they are, by definition, read-only operations; the file or database is not changed. Updating typically requires more control, such as editing and internal consistency checking.

C.3 APPLICATION PACKAGES

Application packages are systems written to handle certain common applications, such as payroll or inventory. Normally, they are parameterized so that each user can tailor the system to his particular specifications. To the extent that a user's requirements are typical of the application, and the application itself is a common one, it is likely that an adequate package will be available. Conversely, highly specialized applications probably should not be implemented with an application package. See [Fran84] for further guidance.

C.4 APPLICATION GENERATORS

Application generators accept some high-level specification of the work to be done and then produce programs (either source or object code) to accomplish that task. If the generated code is in source form, then, of course, the user is free to modify it directly. Thus, there may still be a problem of source code

maintenance (and standardization), unless the user is committed to changing the application only through the generator itself. Generators usually provide an escape mechanism, so that users can code certain crucial parts of the system by hand.

C.5 VERY HIGH-LEVEL LANGUAGES

It is an open question which languages qualify as "very high-level." In [Mart82], the term is applied to APL, NOMAD, and MAPPER, because they provide powerful operations for data manipulation not found in languages such as Pascal or PL/I. Others use the term to refer to more research-oriented languages, such as SETL and FFP [Samm81]. In any event, these languages encourage a more functional (and therefore less procedural) style of programming than do the traditional algorithmic languages.

C.6 ASSEMBLER LANGUAGE

Another alternative, albeit not a new one, to the use of high-level languages is the use of assembler language. There remain applications for which direct control over the hardware or run-time efficiency is so important that assembler programming is justified. We should stress that it is a rare application which must be programmed completely in assembler. More often, a relatively small piece of code accounts for a high percentage of execution time, or is inherently machine-dependent. In such cases, it is reasonable to program the critical section of code in assembler, while most of the system is expressed in some higher-level language. Note also that the "mid-level" language C can often be used in place of true hardware-oriented assembler, with comparable efficiency. C is covered below in detail.

C.7 MANUAL OPERATIONS

There is always some cost associated with automating an application (e.g., for hardware, for the staff time to develop and operate the system). When a system is implemented through conventional programming, this cost is not likely to be negligible. Therefore, it is reasonable to ask whether automating a given application is worth the effort. It is not clear, for instance, that much is gained by having an appointment schedule implemented on a personal computer rather than in a notebook.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** (See instructions)	1. PUBLICATION OR REPORT NO. NBS/SP-500-117/1	2. Performing Organ. Report No.	3. Publication Date October 1984

4. TITLE AND SUBTITLE

Computer Science and Technology:

Selection and Use of General-Purpose Programming Languages--Overview

5. AUTHOR(S)

John V. Cugini

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

Same as in item 6 above.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 84-601119

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

Programming languages have been and will continue to be an important instrument for the automation of a wide variety of functions within industry and the Federal Government. Other instruments, such as program generators, application packages, query languages, and the like, are also available and their use is preferable in some circumstances.

Given that conventional programming is the appropriate technique for a particular application, the choice among the various languages becomes an important issue. There are a great number of selection criteria, not all of which depend directly on the language itself. Broadly speaking, the criteria are based on 1) the language and its implementation, 2) the application to be programmed, and 3) the user's existing facilities and software.

This study presents a survey of selection factors for the major general-purpose languages: Ada, BASIC, C, COBOL, FORTRAN, Pascal, and PL/I. The factors covered include not only the logical operations within each language, but also the advantages and disadvantages stemming from the current computing environment, e.g., software packages, microcomputers, and standards. The criteria associated with the application and the user's facilities are explained. Finally, there is a set of program examples to illustrate the features of the various languages.

This volume contains the discussion of language selection criteria. Volume 2 comprises the program examples.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Ada; alternatives to programming; BASIC; C; COBOL; FORTRAN; Pascal; PL/I; programming language features; programming languages; selection of programming language.

13. AVAILABILITY

[X] Unlimited

☐ For Official Distribution. Do Not Release to NTIS

[X] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

81

15. Price

USCOMM-DC 6043-P80

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

(Notification key N-503)

NBS Technical Publications

Periodicals

Journal of Research—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year.

Nonperiodicals

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396). NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.
Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*
Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.